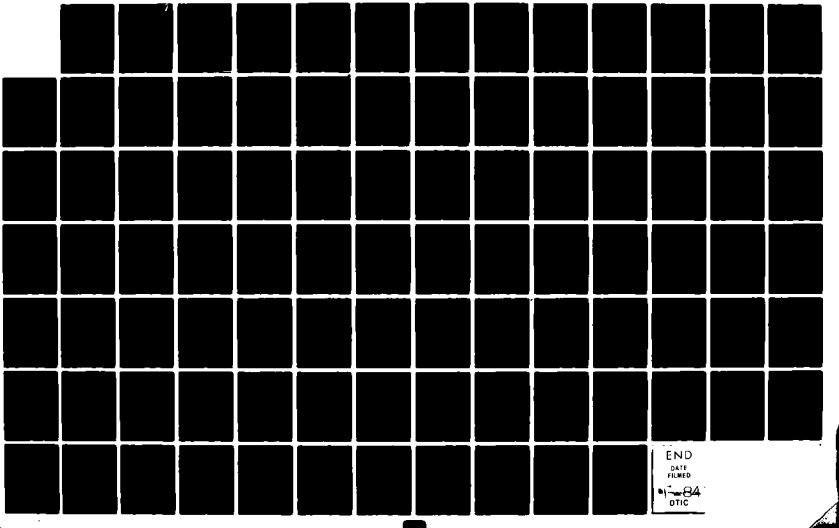
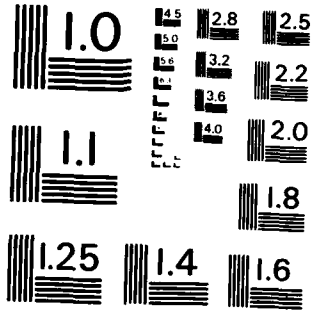
AD-A135 865 AN IMAGE PROCESSING LANGUAGE(U) AIR FORCE INST OF TECH 1/1
WRIGHT-PATTERSON AFB OH R C HOOD DEC 83
AFIT/CI/NR-83-62T

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
1-84
DTIC



AN IMAGE PROCESSING LANGUAGE

By

ROBERT C. HOOD

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1983

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFIT/CI/NR 83-62T	2. GOVT ACCESSION NO: A135865 — 3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): An Image Processing Language	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Robert C. Hood	8. CONTRACT OR GRANT NUMBER(s): -
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: University of Florida	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433	12. REPORT DATE: Dec 1983
	13. NUMBER OF PAGES: 83
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

8 DEC 1983

LYNN E. WOLAVER
Dean for Research and
Professional Development

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN IMAGE PROCESSING LANGUAGE

By

Robert C. Hood

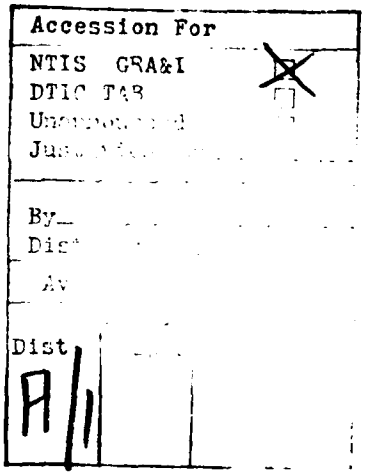
December 1983

Chairman: Leslie H. Oliver, Ph. D.
Major Department: Computer and Information Sciences

This thesis introduces a new image processing language (IPL) which is based on the domains and operations required to express common image processing algorithms in a high-level language. IPL is independent of any particular computer architecture. Its implementation on a computer can take advantage of the machine's architecture without affecting the correctness of algorithms written in it. IPL includes picture, mask, region, boundary, and histogram domains. Unary, binary, and fly operations are described for picture, mask, and region domains and several examples of their use are provided through the expression of several common image processing algorithms in IPL. IPL also includes several set and component selection operations.

The IPL is implemented in Ada* to provide a syntax for its use and a second description of its semantics. The Ada implementation of IPL has not been compiled or tested.

Chairman



* Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my committee chairman, Dr. Leslie H. Oliver, for his unfailing guidance and advice on this study. Special thanks are also due my committee members, Dr. Donald G. Childers and Dr. Sukhamay Kundu, for their patience and participation.

Finally, I offer very special gratitude to my wife and children for their patience and understanding. Without their consideration this effort would never have been undertaken.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN IMAGE PROCESSING LANGUAGE

By

Robert C. Hood

December 1983

Chairman: Leslie H. Oliver, Ph. D.
Major Department: Computer and Information Sciences

This thesis introduces a new image processing language (IPL) which is based on the domains and operations required to express common image processing algorithms in a high-level language. IPL is independent of any particular computer architecture. Its implementation on a computer can take advantage of the machine's architecture without affecting the correctness of algorithms written in it. IPL includes picture, mask, region, boundary, and histogram domains. Unary, binary, and fly operations are described for picture, mask, and region domains and several examples of their use are provided through the expression of several common image processing algorithms in IPL. IPL also includes several set and component selection operations.

CHAPTER ONE
INTRODUCTION

The rapid development of image processing by digital computers in the past decade provides increased emphasis on the definition of machine architectures and programming languages which improve the digital computers' performance in this area. Several workshops have been convened on these topics and the proceedings of the 1980 and 1981 workshops are reported in Duff and Levialdi (1981). In the preface to this report Duff and Levialdi summarize three different approaches taken to develop an image processing language,

> to program in a high-level language (such as, for example Fortran) and to call from an image processing library subroutines optimally designed for a given machine; or to use an interpreter like APL or PICASSO in which interactively and accurate diagnostics may be obtained; or finally, to define a high-level language having specific control structures for local computations and global parallelism, as will as data structures and data types particularly useful in this field. (Duff and Levialdi, 1981 p. vii)

The approach taken in this thesis to develop an image processing language is distinct from these three approaches. Instead the Image Processing Language (IPL) described in this thesis is developed from an examination of the domains and operations which are commonly used to express image processing algorithms in strongly typed, high-level

languages. IPL is focused on abstract data types. The exact structures used to represent a data type and the exact implementation of the operations for these types are omitted whenever possible. This approach removes consideration for a particular architecture and emphasizes the fundamental characteristics of these domains and operations.

Goals

The motivation behind this approach is to develop a readable and portable language. In order for the language to be readable it must be capable of expressing image processing algorithms in a manner which models their use to perform image processing rather than their development or implementation. To be portable the language must separate the expression of algorithms from their implementation.

Four immediate goals which support the developmant of a readable and portable language are to

- develop a language which contains a basic set of operations required to express common image processing operations;
- describe an implementation of the language in an existing, portable language;
- demonstrate the language's capabilities by expressing some common image processing algorithms in it;
- and to provide for the capability to implement the language on different machine architectures.

Of course the efficiency of algorithms is of critical importance to image processing and an algorithm's efficiency is closely coupled to the architecture on which it is

implemented. Therefore, a secondary objective of this language is

- to define the operations required in an architecture designed for image processing.

By expressing algorithms in a manner which models their use rather than their development or implementation, a different insight into the architectural requirements inevitably results. Thus unique architectural ideas may be brought out or the usefulness of previous ideas confirmed.

Previous Work

Even though the approach taken to develop IPL is different from previous approaches, several other languages were studied to insure that previously examined and useful data structures or operations were not omitted. PIXAL (Levialdi et al., 1981), L (Radhakrishnan et al., 1981), PICASSO (Kulpa, 1981), MAC (Douglas, 1981), PPL (Gudmundsson, 1981), QPE (Chang, 1981), AND PLANG (Sinha, 1983). Maggiolo-Schettini (1981) provides a review of these languages and outlines five crucial features used to measure the expresssive power of a language for image processing. These features are:

(a) possibility of defining arrays on which to operate in parallel;

(b) possibility of selecting a subarray of an array for partial processing;

(c) possibility of comparison between the neighborhood of an element of an array and a given pattern;

(d) availability of parallel instructions with global control;

(e) availability of parallel instructions with local control. (p. 159)

All of these features are availbale in the implementation of IPL presented in this thesis. In addition, the connection between computer graphics and image processing languages was examined to insure that any capabilities in this similar field were also included (Foley and Van Dam, 1983; Williams, 1979). No unique contributions from computer graphics languages were found, however.

The majority of the languages for image processing studied are characterized by an emphasis on the implementation hardware available. In addition, they were generally overlayed onto a high-level language such as FORTRAN, PASCAL, or ALGOL, and oriented towards a particular class of digital picture. These characteristics limited these languages portability and readability.

Since a certain amount of sequential programming is required in any language operating on currently available processors the IPL includes sequential constructs which are useful for writing readable and portable code. These constructs are D-type control structures and data type definition capabilities (both domain and operations over the domain). The precise syntax for these capabilities is not described here since they have been extensively studied and described elsewhere, and their precise syntax is not

important at this stage of the language development. Ledgard and Marcotty (1981) provides the definition of these features for this development.

Since the syntax for the D-type structures and data typing are not included here, an existing language was used to provide these syntax during IPL development. Ada* filled this role. It was chosen because it had recently been accepted as an ANSI standard and the development of the language was heavily influenced by requirements for readability and portability (Bulman, 1981), which are the primary goals for IPL. The use of Ada as a starting point for IPL did not restrict its development to only those ideas which could be expressed in Ada, however. The "For Each" control structure described later is the best example of this. In fact, the IPL description is abstracted from any implementation and made in an algebraic description of the domains and operations over them. It is assumed, however, that Ada's array assignment and data type attribute capabilities are available. These capabilities are described in detail in an Ada language description such as Ada Programming Language (1983). The ommission of these features from an IPL implementation would limit the robustness of the language but does not effect its expressiveness.

* Ada is a trademark of the Department of Defense (Ada Joint Program Office).

Organization

This description of IPL is organized into three topics: a description of IPL in mathematical terms; an implementation of IPL in Ada (Ada-IPL); and examples of the use of Ada-IPL to express some common image processing operations. The language description is formed by defining IPL objects' domains in Chapter two. The operations over these domains and a control structure for IPL are defined in Chapter three. The Ada implementation of IPL in Chapter four provides a second description of the language, a syntax for demonstrating its use, as well as a capability to use IPL immediately with an Ada compiler. Finally, the examples of Ada-IPL's use in Chapter six provide an evaluation of IPL's expressiveness and provide a second description of its semantics.

CHAPTER TWO
IPL OBJECTS

Five domains are described in this chapter, picture, mask, region, boundary, and histogram. Objects from these domains are commonly used to perform digital image processing.

Picture

Informally, an image is a two-dimensional object whose brightness or color may vary from point to point and is usually modeled as a real valued function of two variables (Rosenfeld and Kak, 1976). This image is digitized by a sampling process which extracts from the image a discrete set of values ("samples"), and then quantizes the samples to yield values from another discrete set. In IPL, such a digitized image is called a picture. In most practical situations, a sample is the average value of the image over some small area. The areas used to form the sample value are generally the area around each member of a discrete, usually regularly spaced, set of points. Thus, a digitized picture can be thought of as a two dimensional array of digitized, average values. The elements of such a picture are called Pats (Picture Atoms) in IPL.

Pats, or Picture Atoms, are referred to in the current literature as elements, pixels, or pels. Although, as described above, a Pat's value is the digitized, average value of the image over some area, in image processing the Pat is generally assumed to contain the digitized value of the image at a point. This may or may not be a good assumption depending upon the sampling point set and area size. For IPL however, this assumption is used. The primary impact of ths assumption will be discussed below, where the boundary of a region is defined. Thus, in IPL, the Pat is the basic element of a picture and is allowed to have any value, or vector of values, that the user defines. In addition, the user must define the operations such as comparison, addition, or any other operations used in an algorithm for the specified Pat domain.

The location of the sampling points are also a part of the IPL picture. There are two common methods of organizing these sampling points, both of which IPL supports. The most common method uses a regularly spaced, square array of points, i.e., points (m*d, n*d) whose coordinates are multiples of some unit distance d. Such an organization is called orthogonal in IPL. The other method commonly used to organize a picture uses a regular hexagonal array of sample points. This array can be obtained from a square array by shifting the odd-numbered (even) rows d/2 to the right. This organization is called Hex-odd (Hex-even) in IPL.

The domain from which pictures may be selected in IPL is defined using the following domains:

Pat: {the discrete set of values allowed for the image samples}

Organizations: {orthogonal, hex-even, hex-odd}

Index: {Values used to specify a location in or around a picture or mask}

Coordinates (C): {<m,n>|<md+kd,nd> are the coordinates of the samples in the image, m and n ϵ Index, and k is determined as follows -

If organization is orthogonal then k=0;

If organization is hex-even then k=0 when m is odd and k=1/2 when m is even;

If organization is hex-odd then k=1/2 when m is odd and k=0 when m is even.}

The domain for pictures is

Picture (P): $\{\langle S, O_p\rangle | S \epsilon \{p_{ij} | \langle i,j\rangle \epsilon$ Coordinates and there exists a, b, c, and d ϵ Index such that if a<=i<=b and c<=j<=d then $p_{ij} \epsilon$ Pat$\}$ and $O_p \epsilon$ Organizations$\}$.

Thus, in order to specify a picture domain in IPL one must specify its Pat domain, Coordinate domain, and organization. The Coordinate domain is specified by stating the Index domain.

The IPL picture is a two-dimensional array and allows the usual array operations of component selection and assignment. Often, operations on a picture require values which are not in the sampled portion of the image. These values are required when performing operations that use the value of "neighboring" elements such as determining the average of a picture's values within a specified radius of each element. Such operations will not have a complete set of picture values when they are applied to Pat locations which are "near" (within the specified radius) the edge of the picture. Those Pat locations which are outside the sampled area of the image but must have defined values for certain operations are defined to be in the border of the IPL picture.

The values assigned to border elements are usually an identity element for the kind of operation being performed, such as the additive identity (zero) for a linear combination. In general, however, these border elements may take on any Pat value (such as the value of the "nearest" available image element). The normal method of handling the border in digitized image processing is to assign some constant value to it. However, since the border value is usually the additive identity for some operation, it is too restrictive to require it to be a constant in the image. Instead, the border value should be a parameter for any operation requiring it. Also, the border value should not be limited to constant values. Indeed, when working with a

mosaic of several pictures the desired border value may be a Pat value from a neighboring picture. A function description provides the most general form of specifying a border value whether it is a constant or a Pat value from another picture. Thus, in IPL, Border_Value is a function which is a parameter for any operation which might require.

Border_Value: C x P → Pat.

Allowing the Border_Value function to be a parameter for an operation both highlights its importance to the operation and allows a change in the method of computing a border value for a particular operation without affecting other operations.

Mask

An object similar to a picture which is also used in image processing is a mask. Informally, a mask is a neighborhood of some Pat, p_{ij}, where the elements of the mask are typically the neighbors of p_{ij}. A mask is used to define operators on a Pat, p_{ij}, which is called the center of the mask in IPL. These operations calculate a new Pat value for the mask center based on the values of neighboring Pats. The mask defines which Pats affect the calculation by specifying their geometric relationship to the Pat center. Thus a mask must have the same organization as some picture, P, to which it is applied. The elements of a mask may or may not be Pat values. Typically a mask may define the coefficients for

some linear combination of the neighbors of its center, so the values of the elements of the mask are the coefficients, and not the Pat values themselves. A Mask which has elements from a Pat domain is called a Pat_Mask. In IPL, the Pat_Mask is a structural object used by a function to calculate a new value from the values of "nearby" Pats. Thus, the mask operation can be any function which uses the values from a Pat_Mask and returns a Pat value. This allows a wide range of functions, including nonlinear and discontinuous.

Structurally a mask and a picture are identical; the difference between the two objects is in their semantics. The conversion from a mask domain to a picture domain (and vice versa) is straightforward due to this structural equivalence. Thus any operations defined for picture objects can be applied to mask objects once they are converted to pictures. The reason for creating masks as separate objects in IPL is that they are used quite differently from pictures to describe an image processing algorithm. A picture is the object on which an algorithm (operation) acts while a mask is an object which defines the algorithm (operation). Typically a mask will have fewer elements than a picture over which it operates, but this is not a restriction. By separating the uses of picture-like objects into those objects which define an operation (mask) and those which are acted on by the operation (picture) the expression of algorithms is clearer.

The conversion between picture and mask domains is only allowed when they both have elements from the same Pat domain. The Sobel operator described in Chapter 6 has an example of this conversion and its usefulness.

Regions

A segment is another important object in image processing. In IPL an object of type region is used to model a segment. Generally, a segment is a subset of an image that is formed using a rule for the inclusion of points in the segment. Generally, the rules for inclusion of a point in a segment take the form of a range of allowable values (both actual or calculated using a mask). If a point has a value in the range, it is included in the segment.

Another important property of a segment is its connectedness. Although the defintion of a segment does not inherently define this property, many operations on a segment use its connectedness. The connectedness of a segment is defined in terms of the adjacency of two points in the segment. Given a segment, if each point in the segment is adjacent to some other point in the segment, then the segment is connected. Adjacency is generally defined by describing the geometric relationship which exists between two points which are adjacent. In IPL the connectedness property is specified by describing an adjacency function. The adjacency function is then a parameter for any operation

requiring the connectedness property. Although in general any function which maps a pair of coordinates into a Boolean value defines an adjacency rule (C x C → Boolean), in practice only three such functions are commonly used. One, 6-connected, is for hexagonal organizations; and two, 4-connected and 8-connected, are for orthogonal organizations. The set of adjacencies available in IPL is

Adjacencies (A): $\{A_r |$

if $O_p \in$ {hex-even, hex-odd} then A_r =c6

else if O_p = orthogonal then $A_r \in \{c4,c8\}\}$.

The definiton of these adjacencies carries their conventional meaning and their definition can be found in Rosenfeld and Kak (1976, p. 335).

In IPL, an object of type Region is used to model a segment. A region is a subset of the Pats in some picture. The picture used to define a region ia a part of the region object; the picture itself, however, remains distinct from the region. The relationship between a region and the picture from which it is derived can be thought of as making a copy of the picture. This permits changes to the region which do not affect the picture and vice versa, thus many constraints on the permissible operations on regions are avoided. The picture and region objects become independent of each other and users may make changes to either regions or pictures without having to consider their origin or relationship to other regions or pictures. In addition, the region's underlying picture defines a limited domain for

operations between two regions (see Region Operations). The domain of a region R from picture P is formally defined as

Region (R): $\{\langle T_r, P_r, O_r\rangle \mid P_r = \langle S_p, O_p\rangle \in$ Picture, T_r is a subset of S_p, and $O_r = O_p\}$

Boundary

Another object used in image processing which is available in IPL is the boundary of a region. Informally, the boundary of a segment in an image is the line which separates points in the segment from points not in the segment. Since the Pats in IPL are digitized representations of points in the image, a boundary of a region can be represented as a collection of the Pats which are in the region but adjacent to a Pat not in the region (in the region's complement). The assumption that Pats are representaions of points in the image is important to this representation. If the Pats are not good approximation of point values but represent areas of the image instead, a better method of describing a boundary would use different objects. As an example of the kind of object which might be used, a boundary could be represented by a collection of the points in the digitized image which are formed when the areas of four Pats meet. But, since we have assumed Pats to be representations of points in the image, the representation of a boundary by a collection of Pats is the best representation. Thus, the boundary B of region R is

Boundary (B): $\{\langle U_b, R_b, O_b, A_b\rangle |$

$R_b = \langle T_r, P_r, O_r\rangle \in$ Regions, $O_b = O_r$,

if $O_b \in$ {hex-even, hex-odd} then A_b = c6,

if O_b = orthogonal then $A_b \in$ {c4, c8},

and $U_b = \{\langle x,z\rangle | z \in$ Class where for $x \in T_r$

there exists $y \in T_r'$ such that

$A_b(x,y)$ = True$\}\}$

where Class: {boundary, limit} and

T_r' is the complement of T_r.

The {boundary, limit} classifications of the Pats which are on the boundary correspond to Pats which are either on the boundary or to Pats which do not have enough information in the picture that the region is derived from to determine whether the Pat is on the boundary or not. These "limit" Pats are located on the edge of the picture and the values of some of these Pat's adjacent Pats are unknown (Pat b in Figure 1). A limit Pat may be on the boundary of a region or it may not, depending on whether the unknown Pats are in the region or not. Since a particular application may not treat these points the same as points that are on the boundary, this differentiation must be made.

o o x z x o o
o o x x x o o
o o o o o o o

FIGURE 1: Boundary and Limit Classifications. Pats x, and z are in a region. Pats labeled x are on the boundary of the region but Pat z cannot be classified as on the boundary based on the information available in the picture. Therefore Pat z is classified as a limit Pat of the boundary.

Note that the adjacency used to specify a boundary does not define a constraint on the adjacency between elements of the boundary. In fact if 8c defines the boundary members, then these members will be connected under 4c while if 4c defines the members then a connected boundary may be connected only under 8c. (Bear in mind that if two coordinates are connected under 4c they are also connected under 8c but not vice versa) (Rosenfeld and Kak, 1976). The boundary may even have collections which have no two elements adjacent under any of the given rules.

As with the relationship between region and picture, a boundary is defined for a specific region and the creation of a boundary object can be viewed as creating a copy of the underlying region. Thus the values in a region may be changed without affecting any boundary objects that it has defined. The boundary differs from the region in that neither Pat values, Region membership, nor Boundary membership may be changed once a Boundary object has been created. In order to make these kinds of changes, the Region object must be changed and then a boundary of the new region created. These restrictions are required in order to insure that the Boundary object continues to represent a Boundary. Thus, Region membership and Pat values are fixed in a Boundary object.

Histogram

A histogram is an object used in image processing to represent the frequency with which each Pat value occurs in a picture. Since the values allowed for a Pat form a discrete set, then

Histogram (H); {<p,i>|p ∈ Pat, i ∈ Positive Integer; and for all x ∈ Pat, there exists at most one y such that <y, i> ∈ H}.

This definition, however, does not always represent the information desired by a user. For example, when the Pat domain is the same as some computer's domain of approximations to Real numbers the user may not wish to have the number of times each Real value occurs but, instead, the number of times a value occurs in some range of the Pat values. (This domain does form a discrete, albeit large, set). In this case the histogram is generally formed by mapping the allowable Pat values into some smaller discrete set, for example integers within a certain range, and then forming the histogram for these values. Thus the general definition of a histogram in IPL is

Histogram (H): {<e,i>|e ∈ a discrete set (D) and i ∈ Positive Integers, and for each x ∈ D there exists at most one <x,i> ∈ H}.

CHAPTER THREE
IPL OPERATIONS AND THE "FOR EACH" CONTROL STRUCTURE

Several operations which define mappings over the picture, mask, region, and boundary domains are defined in this chapter. Also defined are a collection of operations which construct a registration between two IPL objects. In addition to these operations one control structure, "For Each", is defined. This control structure improves the readability of a sequence of operations which are applied to every member of a set.

Picture Operations

In this section four basic operations on pictures are described. Three of these operations, Picture_Binary_Operations, Picture_Unary_Operations, and Fly have incomplete or generic descriptions which require the user to fill in certain details before they can be used. The other operation, Rotate_90, is completely described. The reason only generic descriptions of the first two operations are given is because most operations on pictures are not closed with respect to a single domain for Pats. In fact, there are very few operations which are performed within a single Pat domain. Typically a picture will enter an algorithm

with a limited Pat domain caused by limited sensor or transmission medium bandwidths. Taking an edge detection algorithm as an example, this picture might then be mapped into a real or integer domain, which would then be mapped into a boolean domain for registration of the edges, and finally mapped into still another limited domain for output (such as to a video display). Clearly it is not possible to define all of these operations without precise descriptions of all the domains and mappings between them. Indeed, these precise definitions are the major effort when defining an image processing algorithm. Thus the generic descriptions which follow provide a framework for specifying the different domains and mappings.

In the descriptions which follow, P1, P2, and P3 represent pictures with Pat values from domains PD1, PD2, and PD3 respectively. PD1, PD2, and PD3 may be distinct or identical domains.

Picture Unary Operations: P1 → P3

The unary operation takes a single picture and maps it into another picture. The mapping is defined by a user defined mapping from PD1 → PD3. Each Pat is operated on independently. P1 and P3 each have the same number of Pats and organization although they do not need to have identical indices. An example of a unary operation is the mapping of one boolean picture into another boolean picture by applying a "not" operation to each individual Pat.

Picture Binary Operation: P1 x P2 → P3

The binary operation takes a two pictures and maps them into another picture. The mapping is defined by a user defined mapping from PD1 x PD2 → PD3. The Pats in P1 are combined with the Pats in P2 which are in a similar location, i. e., the Pat in P1's first row and first column is combined with the Pat in P2's first row and first column to yield the Pat in P3's first row and first column. P1, P2, and P3 all have the same number of Pats and organization although they do not need to have identical indices. An example of a Picture_Binary_Operation is to divide one integer picture by another integer picture to give a real valued picture.

Rotate 90: P1 x Positive Integer → P1

This operation returns a picture with the same number of elements, organization, and Pat domain as the input picture. The operation moves the value in the input picture at location <P1'First(1) + i, P1'First(2) + j> to <P2'First(1) + j, P2'Last(2) - i>. First(1) and First(2) represent the picture's first row and column index value respectively; Last(2) represents the picture's last column index value. This "exchange" is repeated the number of times specified by the positive integer.

Fly: P1 x Mask Operation x Mask x C x Border Value → P3

Masks may be used to transform one picture into another. In IPL this operation is called Fly. The Fly operation computes a new picture from an old picture based on the transformation defined by a Mask_Operation. As for Picture operations, Fly may be defined for pictures with different domains.

The Fly function creates a new picture by computing a new Pat value in the Mask_Operation specified when the Fly function is created. The Mask_Operation is

Mask_Operation: Mask1 x Mask2 → PD3.

Fly fills a mask with Pat values from the input picture P1 which have the same positional relationship with respect to the Pat value being calculated as the mask coordinates do with respect to the input center coordinates (C). The size of this mask is identical to the size of the mask input to Fly. Fly then calls the Mask_Operation with the mask filled from P1 and the mask input to Fly to get a new Pat value. Thus the Mask_Operation is always given masks of equal size. By repeating this procedure for each Pat in the Picture a new picture is generated. Fly also calls a Border_Value function to determine a Pat value when a value outside the picture is required. A similar Fly operation is available for regions with restricted mask domains and operations.

A common Mask_Operation found in image processing is the linear combination of the Pats in the two input masks. This operation is the sum of the elements of the binary

product of the two masks. (The binary product of two masks is equivalent to a Picture_Binary_Operation where the operation is the product of two Pats). The definiton of linear combination in this manner requires that the masks be of the same size, that the product PD1 x PD2 → PD3 is defined, and that addition is defined over PD3. An instance of the function requires the user to specify the domains for the masks and for the output, and to insure that the above operations are defined.

The mask operations can be used to define many common image processing operations such as filtering, differentiation, texture edge enhancement, Sobel edge enhancement, and template matching by cross correlation. Examples of these operations written in the Ada implementation of IPL are given in Chapter 5.

Region Operations

Eleven basic operations are available for a region: Pat_Value, Make_Pat_Region, Make_Region, In_Region, Union, Intersection, Difference, Complement, Picture_Of, Region_Binary_Operation, and Region_Fly. Each operation is described below.

Pat Value: R x C → Pat

This operation returns the value of the Pat located at the given coordinates in the region. A Constraint_Error occurs if the given coordinates are not within the bounds of the region.

Make_Pat_Region: Pat x C → R

This operation creates a region object from a picture object and the coordinates of a Pat in the picture. A Constraint_Error occurs if the given coordinates are not within the bounds of the region.

Make_Region: P x Boolean_Picture → R

This operation creates a region object from a picture object and another picture of equivalent size and index values but with boolean elements (Boolean_Picture). Those elements in the Boolean_Picture with a value of True are placed in the output region object while False locations are not included.

In_Region: R x C → Boolean

In_Region returns True if the given coordinates are in the given region and False otherwise. A Constraint_Error occurs if the given coordinates are not within the bounds of the region.

Union "+": R x R → R

This operation returns the region which contains all Pats in either of the two input regions. The two input regions must be formed from the same picture or an Incompatible_Regions error will occur. The output region will be formed from the input regions' underlying picture.

Intersection "*": R x R → R

This operation returns the region which contains all Pats in both of the two input regions. The two input regions must be formed from the same picture or an Incompatible_Regions error will occur. The output region will be formed from the input regions' underlying picture.

Difference "-": R x R → R

The difference operation returns the region with Pats in the first input region but not in the second input region. The two input regions must be formed from the same picture or an Incompatible_Regions error will occur. The output region will be formed from the input regions' underlying picture.

Complement: R → R

This operation returns a region over the same picture as the input region but whose members are not in the input region.

Picture Of: R → P

This operation returns the underlying picture of the given region.

Region Binary Operation: R x P → R

The Region_Binary_Operation provides the capability to combine a region object with a picture object to form a new region. The inputs must be of the same size or a Constraint_Error will occur. The output region will be

formed from the input region's underlying picture. Membership in the output region is determined by applying a user supplied function which maps

Boolean x Pat → Boolean.

The function is applied to similarly located values of the two inputs just as in Picture_Binary_Operations.

Region Fly: R x BMO x Mask x C x Border Value → R

The Region_Fly operation provides the capabilty to calculate a new region based on whether nearby Pats are in the region or not. The Boolean_Mask_Operation (BMO) supplied by the user provides this mapping and is of the form

Boolean_Mask x Mask → Boolean.

This operation calls a Border_Value function, just as the Fly operation did, to determine region membership values for Pats which are not in the picture used to define the region but whose values are required for Fly to complete. The output region will have the same underlying picture as the input region and will have members which have True results from the BMO. The Boolean_Mask will be filled by the Region_Fly operation similar to the manner used in the Fly operation to fill its Pat_Mask.

Boundary Operations

The basic operations available for a boundary are: Pat_Value, In_Region, Make_Boundary, On_Boundary, Next_Pat, and Region_Of. These 5 basic operations can be used to

define other operations on boundaries such as length, slope, or curvature. In addition, a boundary can be converted to a region with the same elements and then any of the region operations can be applied to the equivalent region object.

Pat Value: B x C → Pat

This operation returns the value of the Pat located at the given coordinates in the picture from the region used to define the boundary. If the coordinates are not within the picture then a Constraint_Error will occur.

In Region: B x C → Boolean

In_Region returns True if the given coordinates are within the region used to define the boundary. If the coordinates are not within the bounds of the region then a Constraint_Error will occur.

Make Boundary (MB): R x A → B

This operation creates a boundary object from the given region using the specified adjacency and the definition of a boundary.

On Boundary: B x C → Class

On_Boundary returns the classification of the Pat located at the given coordinates within the boundary. A Constraint_Error occurs whenever the coordinates are not within the bounds of the underlying region.

Next Pat: B x C x C x A x Direction → C, Where
Direction {Clockwise, Counterclockwise}

Next_Pat provides a method of traversing a boundary by stepping between Pats which are on the boundary or are limit Pats. Steps are made only between Pats which are adjacent under the given adjacency rule. An algorithm for performing this border following is described in detail in Rosenfeld and Kak (1976 p. 342). The direction indicates whether the next step is to be taken in a clockwise or counterclockwise direction. Two coordinates are required; one designates the Pat from which the step is to be made while the other indicates the Pat from which the search for the next Pat is to be initiated. The later Pat must be at least c8 adjacent (c6 for Hex organizations) to the former Pat. The Pat from which the step is to be made must be either on the boundary or a limit Pat. The Pat from which the search is initiated may be of any classification. (The Pat visited one step back guarantees that the boundary will be traversed completely.)

The border following algorithm used has no memory of which Pats have been visited but merely calculates the next Pat based on the given information. Thus this operation will always return the coordinates of a Pat classified as On or Limit. Users must remember where they have been and decide when they have visited every Pat on the given boundary. Some boundaries may have Pats that will be visited as many as three times before all Pats have been

visited. In addition, this operation will not jump to disconnected boundary segments within the same boundary object. Finally, the operation visits only the minimum number of Pats required to traverse the given boundary under the given adjacency. It is possible that some Pats may not be visited. For example, if the boundary was created with a c8 adjacency rule and then traversed with the same rule, not every member of the boundary must be visited to define a connected path. Figure 2 shows an example of this situation.

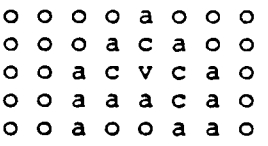


FIGURE 2: Border Following Around a Closed Boundary. In this picture Pats labeled as a, c, or v are in a region. Pats labeled as a are on the c4 boundary while Pats labeled as a or c are on the c8 boundary. a traversal of the c8 boundary using steps between Pats which are c8 adjacent would visit only the a Pats. In order to visit both the a and c Pats the boundary must be traversed under c4.

Region Of: B → R

This operation returns the underlying region of the boundary.

Registration

In image processing it is often necessary to define a relationship between two images. This relationship generally defines the same or similar objects within the two images. The objects within the two images may be identical or they may be modified by a rotation or scale of the object. In any case this relationship is a mapping between the points of the two images. In IPL, such a mapping is called a registration (Rg).

$Rg(p_{ij}) \rightarrow p_{mn}$, where $p_{ij} \in P_a$, $p_{mn} \in P_b$,
and picture P_a is registered to picture P_b.

A registration can be defined by either a functional mapping of the coordinates of the first picture into the coordinates of the second picture, by an enumeration of the points in the first picture and their associated points in the second picture, or by some combination of the two.

In order to define the registration of two pictures by an enumerated set this object and its associated operations must be defined. The object is

Set_of_Coordinate_Pairs (SCP): {<a,b>|
a,b ∈ Coordinates; a is the coordinate
of a Pat in P_a to which b, the coordinates
of a Pat in P_b is registered; and
for each x ∈ {the coordinates of a Pat in P_a}
there is at most one <x,b> ∈ SCP}.

The last requirement of a Set_of_Coordinate_Pairs element allows the first coordinate in a coordinate pair to be an

index for that element and is the basis for the Registered_Coordinates use of the set. A Set_of_Coordinate_Pairs has the operations listed in the following sections.

Make Coordinate Pair Set: C x C → SCP

This operation creates a SCP object from two coordinates. The coordinates must designate Pats of the pictures which defined the SCP or a Constraint_Error occurs.

Union "+": SCP x SCP → SCP

This operation returns the SCP which contains all registered coordinates in either of the two input SCPs. If <a,x> is in one of the input SCPs and <a,y> is in the other, then x = y or an Inconsistent_Registration error occurs. An Inconsistent_SCP error occurs if the two input SCPs are not defined over the same pictures.

Difference "-": SCP x SCP → SCP

The difference operation returns the SCP with registered coordinates in the first input SCP but not in the second SCP. If <a,x> is in one of the input SCPs and <a,y> is in the other, then x = y or an Inconsistent_Registration error occurs. An Inconsistent_SCP error occurs if the two input SCPs are not defined over the same pictures.

Intersection "*": SCP x SCP → SCP

This operation returns the SCP which contains all registered coordinates in both of the two input SCPs. If <a,x> is in one of the input SCPs and <a,y> is in the other, then either x = y and <a,x> will be in the output SCP or an Inconsistent_Registration error occurs. An Inconsistent_SCP error occurs if the two input SCPs are not defined over the same pictures.

Registered Coordinates: SCP x C → C

This operation returns the coordinates which are registered to the input coordinates in the input SCP. A Constraint_Error occurs if the input coordinates do not designate a Pat of the from-picture (P_a) of the SCP.

Change Registered Coordinates: SCP x C x C → SCP

This operation changes the pair of coordinates in the input SCP to the pair designated by the input coordinates. If <a,b> represents the input coordinate pair then a must designate a Pat in the from-picture (P_a) and b must designate a Pat in the to-picture (P_b) or a Constraint_Error will occur. This operation can be defined in terms of the previously defined operations.

"For Each" Control Structure

In adddition to the basic control structures available in most high level programming languages, and specifically those available in Ada, one additional control structure for IPL is desirable. This control structure defines a loop to be repeated for each element of a set. Certainly this control structure can be expressed using "D" type structures and will be defined using them:

```
element := first element the set might contain;
Done := False;
While Not Done Loop
    If element in set Then
        Block of statements operating on element
    EndIf
    If element is the last one in the set Then
        Done := True
        Else element := next element which might
            be in the set
    EndIf;
EndLoop;.
```

The "For Each" control structure is equivalent to the above statements but reduces the code required and the potential for making a mistake in the code. The syntax of the For Each statement is

```
For Each <identifier> of <set_identifier> Loop
    statement...
EndLoop.
```

CHAPTER FOUR
ADA IMPLEMENTATION

Appendices A and B contain the Ada generic package for an implementation of the IPL language. Appendix A contains the package declarations; appendix B contains the package body. Ada was used for this implementation because of its generic program units and its capability to package together data type descriptions with their defined operations while leaving implementation details hidden. The generic capability permits the definition of a program template from which (nongeneric) program units can be obtained. This permits the creation of a program which can operate on a class of data types without writing a separate program for each distinct type. The picture binary and unary operations are excellent examples of generic functions. These two functions operate on pictures, a class of two dimensional arrays with unspecified individual elements. An actual binary function can be created by specifying the kind of elements in the picture and the way in which these elements are to combined. The Ada generic unit also provides an indirect method for passing a function name as a parameter to a function. This capability is used in almost all of the generic units but was the primary reason for making the Fly

functions generic. The generic capabiltiy of Ada was most useful in creating this implementation of IPL. In fact, the entire implementaion is a generic package which can create several different image processing packages for distinct Pat domains. Binary, unary, and Fly operations can then be specified to define how these different domains can be mapped to one another, if at all.

It is also important to note that Ada effectively hides the implementations of the various IPL operations from users resulting in two advantages. First, a user of the language can write in IPL by using the declaration section of the IPL generic package for the syntax of the language, and the language description as the semantics of the language. This contributes to the portability of the code by inhibiting the use of implementation details. Second, the implementations of the operations and the private data types (regions and boundaries) can be changed without affecting the correctness of algorithms written in the language as long as the algorithms do not rely on side effects for correctness and the new implementations conform to the IPL semantics. Thus the given implementations could easily be changed or modified to take adavantage of hardware capabilities, such as assigning multiple processors to the unary and binary operations.

Additionally, the IPL language requires the availability of D-type control structures and the capabilty to specify data type domains as well as operations over

those domains. By embedding IPL within Ada these requirements are met. A user of the language has full access to the Ada programming language as well as the specialized constructs built for image processing. Thus the image processing algorithms presented in this thesis use several of the Ada capabilites.

The Ada Programming Language, ANSI/MIL-STD-1815A (1983), is the source of information used to write this implementation of IPL in Ada. No compiler was available so the given implementation has not been compiled or tested. It is included here as a second description of the language and as a vehicle to demonstrate the usefulness of IPL in stating image processing algorithms. There are two additional benefits, however. First, given an Ada compiler the testing and use of the Ada IPL is straightforward. Since the Ada language is intended to be transportable, this contributes to a transportable IPL. Second, the implementation of IPL in other languages can be modeled directly from the Ada implementation. Ambiguities caused by translating the Ada implementation can be resolved by referring to the more formal description presented in this thesis.

In order to implement the "For Each" control structure, an additional function is added to the Ada IPL. Since a new control structure could not be added directly to the language without writing a precompiler or altering the existing compiler, a function was added which would return a

"next element" for those sets which required it. To step through the Pats in a picture is a direct use of the Ada "For" loops and does not require a "Next" function. Also, the Boundary object already has a "Next" function (border following) which can be used to step through the Pats in a boundary. Regions and Sets_of_Coordinate_Pairs, however, require a "Next" function in order to perform the "For Each" control. The region and Set_of_Coordinate_Pairs "Next" functions are given coordinates to mark where they are in the set and then return the next coordinates in the set. Null coordinates are returned at the end of the set and null coordinate inputs request the first element in the set. These functions proceed through the elements in the set in an unspecified order without repetition.

Finally, the bodies for the procedures to input or output a picture are not included in the body of the IPL package. The bodies of these operations are entirely dependent upon the hardware and architecture of the implementation and therefore cannot be defined here.

CHAPTER FIVE
ALGORITHMS IMPLEMENTED IN IPL

Several algorithms which perform common image processing tasks are written in the Ada-IPL and presented in this chapter. The algorithms demonstrate IPL's capability to express them clearly and completely. At the same time these algorithms demonstrate how IPL highlights the design of the domain mappings critical to most image processing. The algorithms presented were selected based on their common usage in image processing and on their capability to demonstrate the use of different IPL operations. Thus readers who are familiar with image processing techniques should already understand the intent of these operations and be able to study the influence of IPL on their clarity. Thresholding, Laplacian, differentiation, texture edge detection, template matching by cross correlation, linear and nonlinear filtering, and Sobel edge enhancement algorithms are presented for picture operations. Expand, shrink, and area algorithms are presented for regions.

Picture Algorithms

Threshold

A threshold algorithm for pictures is presented in Figure 3 (Rosenfeld and Kak, 1976, p. 258,; Pratt, 1978 p. 534). The algorithm is basicly a unary operation on the input picture which applies a threshold function to each Pat in the picture. By writing the Pat_Threshold function within the declaration section of Picture_Threshold the additional parameters, threshold_values and output_values, are visible to Pat_Threshold. This allows the Pat_Threshold function to conform to the parameter list required by Picture_Unary_Operation. An implementation of IPL which allows parameter lists as well as function names to be parameters would simplify this algorithm.

```
FUNCTION Picture_Threshold (P : Pictures;
                    Lower_Bound, Upper_Bound,
                    In_Range_Value, Out_Range_Value : Pats)
      RETURN Pictures is
    FUNCTION Threshold (Pat : Pats) RETURN Pats is
      BEGIN
        IF (Pat >= Lower_Bound) OR (Pat <= Upper_Bound)
          THEN RETURN In_Range_Value;
          ELSE RETURN Out_Range_Value;
          ENDIF;
        END Threshold;
    FUNCTION Threshold_Op is NEW Picture_Unary_Op
                    (Element_1 | Element_3 => Pats,
                     Picture_1 | Picture_3 => Pictures,
                     F => Threshold);
  BEGIN
    RETURN Threshold_Op(P);
    END Picture_Threshold;
```

FIGURE 3: Ada-IPL Threshold Algorithm.

Laplacian

The Laplacian is a higher-order derivative operator which is used to detect edges but is less sensitive to orientation than a straight derivative operator (Rosenfeld and Kak, 1976, p. 28,1; Pratt, 1978 p. 482). A Lapalacian operation over a digital picture can be defined using a linear combination of floating point and Pat values. Figure 4 contains the declarations needed for such an operation. Using these declarations, the Laplacian operation could be expressed as

```
Float_Pic := Float_LC_Fly (Pat_Pic, ((0,  1, 0)
                                     (1, -4, 1)
                                     (0,  1, 0)), (2, 2));.
```

Furthermore, the Float_LC_Fly can be used to express any linear combination of floating point and Pat values over whatever mask size required.

```
TYPE Float_Mask is ARRAY
    (Mask_Index RANGE 1..2, Mask_Index RANGE 1..2) of Float;
TYPE Float_Picture is ARRAY
        (Index RANGE <>, Index RANGE <>) of Float;
FUNCTION Float_LC is NEW Linear_Combination
                      (Element_1 => Pats,
                       Mask_1 => Pat_Mask,
                       Element_2 | Element_3 => Float,
                       Mask_2 => Float_Mask);
FUNCTION Float_LC_fly is NEW Fly
    (Mask_Index => Mask_Index,
     Element_1 => Float, Mask_1 => Float_Mask,
     Element_2 => Pats, Mask_2 => Pat_Mask,
     Picture_2 => Pictures,
     Element_3 => Float, Picture_3 => Float_Picture,
     F => Float_LC);
```

FIGURE 4: Ada-IPL Laplacian Declarations.

Differentiation

The derivative of a function of two variables can be characterized by its gradient. The gradient is described by its magnitude and direction (Rosenfeld and Kak, 1976, p. 278). Figure 5, p. 42, presents an Ada-IPL algorithm for computing the derivative of a picture. The derivative is computed by first finding the difference between adjacent Pats in the rows and columns. These are computed separately using fly operations. These two pictures are then combined in a Picture_Binary_Op to form a picture with gradients as its elements' values.

Texture Edge Detection

The algorithm described in this section is used to detect the edges around objects which differ from their background with respect to the average value of some local property as opposed to objects which are characterized by some Pat value. The pictures on which this algorithm would be useful generally have a large amount of salt and pepper noise, or noise at both extremes of the Pat domain (Rosenfeld and Kak, 1976, p. 294). The algorithm is shown in Figure 6, p. 43. The algorithm uses a variable size mask which depends on the size, or radius, used to compute the average Pat value. It then proceeds in a manner similar to the differentiate algorithm, only it finds the differences between the centers of the masks used to compute the Pat averages rather than the differences between adjacent Pats.

```
TYPE Gradients is RECORD
       Magnitude, Direction : Float;
       END RECORD;
TYPE Derivative_Pictures is ARRAY
           (Index RANGE <>, Index RANGE <>) OF Gradients;
FUNCTION Differentiate_Picture (P : Pictures)
                        RETURN Derivative_Pictures is
    TYPE Float_Mask is ARRAY
     (Mask_Index RANGE 1..2, Mask_Index RANGE 1..2) of Float;
    TYPE Float_Picture is ARRAY
           (Index RANGE <>, Index RANGE <>) of Float;
    FUNCTION Float_LC is NEW Linear_Combination
                           (Element_1 => Pats,
                            Mask_1 => Pat_Mask,
                            Element_2 | Element_3 => Float,
                            Mask_2 => Float_Mask);
    FUNCTION Float_LC_fly is NEW Fly
                           (Mask_Index => Mask_Index,
                            Element_1 => Float,
                            Mask_1 => Float_Mask,
                            Element_2 => Pats,
                            Mask_2 => Pat_Mask,
                            Picture_2 => Pictures,
                            Element_3 => Float,
                            Picture_3 => Float_Picture,
                            F => Float_LC);
    FUNCTION Derivative_Op (Left, Right : Float)
                          RETURN Gradients is
        Dummy : Gradients;
      BEGIN
        Dummy.Magnitude := Sqrrt(Left**2 + Right**2);
        Dummy.Direction := Arctan(Left/Right);
        RETURN Dummy;
        END Derivative_Op;

    FUNCTION Pic_Derivative_Op is NEW Picture_Binary_Op
                 (Element_1 | Element_2 => Float,
                  Picture_1 | Picture_2 => Float_Pictures,
                  Element_3 => Gradients,
                  Picture_3 => Derivative_Picture,
                  F => Derivative_Op);
    Del_X,
    Del_Y : Derivative_Pictures(P'RANGE(1), P'RANGE(2));
  BEGIN
    Del_X := Float_LC_Fly (P, ((1, -1),
                               (0, 0)), (1, 1));
    Del_Y := Float_LC_Fly (P, ((1, 0),
                               (-1, 0)), (1, 1));
    RETURN Picture_Derivative_Op (Del_X, Del_Y);
    END Picture_Derivative;
```

FIGURE 5: Ada-IPL Differentiate Operation

```
FUNCTION Texture_Edge_Detection
          (P : Pictures;
           R : Index RANGE 1..Index'LAST)     -- Radius
       RETURN Pictures is
-- Assumes Pats is a subtype of Natural Integers
    FUNCTION Integer_LC is NEW Linear_Combination
          (Element_2 => Pats, Mask_2 => Pat_Mask,
           Element_1 | Element_3 => Integer,
           Mask_1 => Mask);
    FUNCTION Integer_LC_Fly is NEW Fly
      (Mask_Index => Index,
       Element_1 | Element_3=>Integer, Mask_1=>Integer_Mask,
       Element_2 => Pats, Mask_2 => Pat_Mask,
       Picture_2 => Pictures, Picture_3 => Integer_Pictures,
       F => Integer_LC);
    FUNCTION "/" is NEW Picture_Binary_Op
     (Element_1 | Element_2 | Element_3 => Integer,
      Picture_1 | Picture_2 | Picture_3 => Integer_Pictures,
      F => "/");
    FUNCTION "+" is NEW Picture_Binary_Op
         (Element_1 | Element_2 | Element_3 => Pats,
          Picture_1 | Picture_2 | Picture_3 => Pictures,
          F => "+");
    FUNCTION Abs (Left : Integer) RETURN Pats is
      BEGIN
        RETURN Pats'Abs(Left);
      END Abs;
    FUNCTION Abs is NEW Picture_Unary_Op
      (Element_1 => Integer, Picture_1 => Integer_Pictures,
       Element_3 => Pats, Picture_3 => Pictures,
       F => Abs);
    Texture_Mask : CONSTANT Integer_Mask
        (-R..R, -R..R) := (OTHERS =>(OTHERS => 1));
    Horiz_Difference_Mask : CONSTANT Integer_Mask
     (0..0, -R..R) := (0 =>(-R => 1, R => -1, OTHERS => 0));
    Vert_Difference_Mask : CONSTANT Integer_Mask
     (-R..R, 0..0) := (-R => (1), R => (-1), OTHERS => (0));
    Normalizing_Picture : CONSTANT
                  Integer_Pictures (P'RANGE(1), P'RANGE(2))
                  := (OTHERS => (OTHERS => (2R + 1)**2));
    Dummy, Horiz_Diff_Pic, Vert_Diff_Pic : Integer_Pictures;
  BEGIN
    Dummy := Integer_LC_Fly (P, Texture_Mask, (0, 0));
    Horiz_Diff_Pic := Integer_LC_Fly
              (Dummy, Horiz_Difference_Mask, (0, 0));
    Horiz_Diff_Pic := Horiz_Diff_Pic/Normalizing_Picture;
    Vert_Diff_Pic := Integer_LC_Fly
              (Dummy, Vert_Difference_Mask, (0, 0));
    Vert_Diff_Pic := Vert_Diff_Pic/Normalizing_Picture;
    RETURN (Abs (Horiz_Diff_Pic)) + (Abs (Vert_Diff_Pic));
    END Texture_Edge_Detection;
```

FIGURE 6: Ada-IPL Texture Edge Detection.

Finally, instead of computing the gradient, it computes an approximation to the magnitude of the derivative by summing the absolute values of the row and column differences.

An important problem for this algorithm is the mapping of Pat values into other domains. This algorithm assumes that the Pat domain is some subset of the Natural integers (0 to +infinity). Then, by using the fact that the Natural integers are closed with respect to addition, it follows that by integer-dividing the sum by the number of elements in the summation, an integer within the Pat subrange will result. Thus the "average" operation is closed with respect to the Pat domain but must be computed with values outside of the domain. In addition, the absolute value function is redefined to return an object of type Pat. In summary then, "/" operates over Naturals, Abs operates on Naturals returning Pats, and there are two "+" operations, one operating over Pats and another (within the linear combination) operating on a Pat and a Natural returning a Natural. Once these mappings are defined, the algorithm follows directly from the mathematical model.

Observe that when using IPL to express such an algorithm as the texture edge detection, the bulk of the effort is to map the Pats domain into a suitable domain(s) for the desired operations, and then to map from that domain(s) back into the Pat domain. Once this is accomplished, it is fairly easy to express the desired mathematical relationships.

Template Matching

The normalized cross correlation can be used to locate areas of a picture which match a template (Rosenfeld and Kak, 1976, p. 298,; Pratt, 1978 p. 551). The method is not described in detail here but in general is accomplished by cross correlating the template with the picture and then normalizing this result by dividing by the square of the picture's Pat values summed over the template area. The algorithm in Figure 7, p. 46, performs this operation for a template of any size. The algorithm includes a straightforward use of the Fly, Binary, and Unary picture operations. Calculations are performed in the floating point domain primarily due to the division required to normalize the cross correlation. There are usually small differences between "exact matches" and "near misses" requiring the increased accuracy of the floating point domain. Although the output of this operation is a picture, it would most likely be next operated on by a search operation to create a registration between the template and the picture. After the completion of this registration, the output picture would most likely be discarded.

Linear Filter

The template matching algorithm above restricted template values to the Pat domain. In general, however, the exact grey levels of a pattern are not as important as the shape of the pattern. Thus it is desirable to cross

```
FUNCTION Normalized_Cross_Correlate
                    (P : Pictures;
                     Template : Pat_Mask;
                     Template_Center : Valid_Coordinates)
                RETURN Float_Pictures is
    FUNCTION Pat_LC is NEW Linear_Combination
                           (Element_1 | Element_2 => Pats,
                            Mask_1 | Mask_2 => Pat_Mask,
                            Element_3 => Float);
-- Must have "*" (Left, Right : Pats) RETURN Float defined
    FUNCTION Pat_LC_fly is NEW Fly
                           (Mask_Index => Mask_Index,
                            Element_1 => Pats,
                            Mask_1 => Pat_Mask,
                            Element_2 => Pats,
                            Mask_2 => Pat_Mask,
                            Picture_2 => Pictures,
                            Element_3 => Float,
                            Picture_3 => Float_Picture,
                            F => Pat_LC);
    FUNCTION "/" is NEW Picture_Binary_Op
       (Element_1 | Element_2 | Element_3 => Float,
        Picture_1 | Picture_2 | Picture_3 => Float_Pictures,
        F => "/");
    FUNCTION "**" is NEW Picture_Unary_Op
                (Element_1 | Element_3 => Pats,
                 Picture_1 | Picture_3 => Pictures,
                 F => "**");
    Correlation_of_Template_and_P,
       P_Squared : Float_Pictures (P'RANGE(1), P'RANGE(2));
    Temp : Pictures (P'RANGE(1), P'RANGE(2));
    Template_of_1s : CONSTANT
            Pat_Mask (Template'RANGE(1), Template'RANGE(2))
            := (OTHERS => (OTHERS => 1));
  BEGIN
    Correlation_of_Template_and_P :=
              Pat_LC_Fly (P, Template, Template_Center);
    Temp := P**2;
    P_Squared := Pat_LC_Fly
                  (Temp, Template_of_1s, Template_Center);
    RETURN Correlation_of_Template_and_P / P_Squared;
    END Normalized_Cross_Correlate;
```

FIGURE 7: Ada-IPL Normalized Cross Correlation

correlate derivatives (magnitude only) of the template with the picture (Rosenfeld and Kak, 1976, p. 306). The declarations in Figure 8 provide the capability to apply a general linear filter to a picture by cross correlating a template of floating point values with it. The template may be any size. The operation of multiplying a Pat and a floating point value to yield a floating point value must be defined for these declarations to be useful.

```
FUNCTION Linear_Filter is NEW Linear_Combination
                         (Element_1 => Pats,
                          Mask_1 => Pat_Mask,
                          Element_2 | Element_3 => Float,
                          Mask_2 => Float_Mask);
FUNCTION Linear_Filter_fly is NEW Fly
                         (Mask_Index => Mask_Index,
                          Element_1 => Float,
                          Mask_1 => Float_Mask,
                          Element_2 => Pats,
                          Mask_2 => Pat_Mask,
                          Picture_2 => Pictures,
                          Element_3 => Float,
                          Picture_3 => Float_Picture,
                          F => Linear_Filter);
```

FIGURE 8: Ada-IPL Declarations for a Linear Filter.

As an example of the use of this operation, suppose we wished to locate thin (one Pat), vertical lines in a picture. A filter for such an operation is the digital Laplacian of an ideal thin vertical line (Rosenfeld and Kak, 1976, p. 306)

```
-1/2   1   -1/2
-1/2   1   -1/2
-1/2   1   -1/2.
```

This filtering can be performed in the Ada-IPL by

```
Filtered_Pic := Linear_Filter_Fly
                (P, ((-0.5, +1.0, -0.5),
                     (-0.5, +1.0, -0.5),
                     (-0.5, +1.0, -0.5)), (2, 2));.
```

Nonlinear Filter for a Thin, Vertical Line

Rosenfeld and Kak (1976, p. 307) point out that the linear filter for a thin, vertical line discussed above has undesirable responses to step edges and isolated points. They then describe a nonlinear filter for detecting these lines (1976 p. 309). Figure 9 displays the declarations required for this filter. An application of this function does not use the element values of an input template since it is tailored for a thin, vertical line only. The fly operation does require a 3x3 mask, however, so that the proper sized Pat mask will be supplied to the Nonlinear_Vert_Line_Filter function. Thus the Fly M1 parameter is provided with a 3x3 mask with "0" elements. The function can be applied to a picture, P, by

```
New_P := Nonlinear_VL_Filter_Fly
                (P, ((0, 0, 0),
                     (0, 0, 0),
                     (0, 0, 0)), (2,2));.
```

```
FUNCTION Nonlinear_Vert_Line_Filter (M1 : Pat_Mask;
                    M2 : Float_Mask)
             RETURN Float is
    FUNCTION Linear_Filter is NEW Linear_Combination
                          (Element_1 => Pats,
                           Mask_1 => Pat_Mask,
                           Element_2 | Element_3 => Float,
                           Mask_2 => Float_Mask);
    I, J : Mask_Index;
  BEGIN
    I := M1'FIRST(1);
    J := M1'FIRST(2);
    IF     M1(I-1, J)   > M1(I-1, J-1)
       AND M1(I-1, J-1) > M1(I-1, J+1)
       AND M1(I, J)     > M1(I, J-1)
       AND M1(I, J-1)   > M1(I, J+1)
       AND M1(I+1, J)   > M1(I+1, J-1)
       AND M1(I+1, J-1) > M1(I+1, J+1)
      THEN RETURN Linear_Filter
                    (M1, ((-0.5, +1.0, -0.5),
                          (-0.5, +1.0, -0.5),
                          (-0.5, +1.0, -0.5)), (2, 2));
      ELSE RETURN 0;
      END IF;
    END Nonlinear_Filter;

SUBTYPE Float_Mask_3x3 is Float_Mask (1..3, 1..3);
FUNCTION Nonlinear_VL_Filter_Fly is NEW Fly
                          (Mask_Index => Mask_Index,
                           Element_1 => Float,
                           Mask_1 => Float_Mask_3x3,
                           Element_2 => Pats,
                           Mask_2 => Pat_Mask,
                           Picture_2 => Pictures,
                           Element_3 => Float,
                           Picture_3 => Float_Picture,
                           F => Nonlinear_Vert_Line_Filter);
```

FIGURE 9: Ada-IPL Declarations for a Nonlinear Filter of a Thin Vertical Line.

```
FUNCTION Sobel (M1 : Float_Mask; M2 : Pat_Mask)
         RETURN Float is
    FUNCTION "*" is NEW Pictur_Binary_Op
        (Element_1 | Element_3 => Float, Element_2 => Pats,
         Picture_1 | Picture_3 => Float_Pictures,
         Picture_3 => Pictures);
-- The Function "*" (Left : Float; Right : Pat)
-- RETURN Float must be defined.
    I : Index;
    X, Y : Float := 0;
    Temp_Mask : Float_Mask (M1'RANGE(1), M2'RANGE(2));
  BEGIN
    Temp_Mask := Float_Pictures(M1) * Pictures(M2);
      -- Masks can be explicitly converted to Picture types
    FOR I IN Temp_Mask'RANGE(1) LOOP
      Y := Y + Temp_Mask(I, 1) - Temp_Mask(I, 3);
      END LOOP;
    FOR I IN Temp_Mask'RANGE(2) LOOP
      X := X + Temp_Mask(3, I) - Temp_Mask(1, I);
      END LOOP;
    RETURN SQRRT (X**2 + Y**2);
    END Sobel;
FUNCTION Sobel_Fly is NEW FLY
                          (Mask_Index => Mask_Index,
                           Element_1 => Float,
                           Mask_1 => Float_Mask,
                           Element_2 => Pats,
                           Mask_2 => Pat_Mask,
                           Picture_2 => Pictures,
                           Element_3 => Float,
                           Picture_3 => Float_Picture,
                           F => Sobel);
```

FIGURE 10: Ada-IPL Declarations for Sobel Edge Enhancement.

Sobel Edge Enhancement

The Sobel operator is a nonlinear edge enhancement technique using a 3x3 mask (Pratt, 1978, p. 487). Figure 10 contains the implementation of this operator in Ada-IPL. As with the nonlinear vertical line filter, the input mask to the fly operation defines the size of the mask required by the Sobel function. However, because of the symmetry for

the coefficients in the Sobel operation the mask elements can be used to pass these values adding another degree of variability to the function. An example of the use of this Sobel edge enhancement is

```
Float_Picture := Sobel_Edge_Enhance
                    (P, ((1, 2, 1)
                         (2, 0, 2),
                         (1, 2, 1)), (2, 2));.
```

Note that a Picture_Binary_Operation is defined for this function and then applied to the masks. Ada provides for explicit conversions between different array types when they have the same dimensionality, index types, and component types (Ada Programming Language, 1983, p. 4-22).

Region Algorithms

Expand

The expand operation is commonly used to expand the region uniformly in all directions towards the edge of the picture. This operation is referred to by Rosenfeld and Kak as propogation (1976, p. 362). Pats are included in the new region if they are in the original region or if they are adjacent to a Pat in the original region. Those Pats in the original region remain in the expanded region. The Ada-IPL implementation of expand is presented in Figure 11. The adjacency mask function in the expand operation is designed to return True if the two input masks have True values in any identical locations and False otherwise. Thus, the body of the algorithm flies the appropriate mask over the picture

```
FUNCTION Expand (R : Region;
                 A : Adjacencies;
                 Num_Times : Integer := 1);
          RETURN Region is
    FUNCTION Adjacency_Mask (M1, M2 : Boolean_Mask)
                      RETURN Boolean is
        I : Index;
        Temp : Boolean := False;
      BEGIN
        FOR I in M1'RANGE(1) LOOP
          M1(1) := (M1(I) AND M2(I)) OR M1(1);
          END LOOP;
        FOR I IN M1'RANGE(2) LOOP
          Temp := Temp OR M1(1, I);
          END LOOP;
        RETURN Temp;
        END Adjacency_Mask;
    FUNCTION Adjacency_Fly is NEW Region_Fly
         (Mask_Index => Mask_Index, Element_1 => Boolean,
          Mask_1 => Boolean_Mask, F => Adjacency_Mask);
    X : CONSTANT Boolean := True;
    O : CONSTANT Boolean := False;
    Dummy : Region (R.Row_First, R.Row_Last,
                    R.Column_First, R.Column_Last);
  BEGIN
    WHILE Num_Times /= 0 LOOP
     IF Organization = Hex_Even
       THEN Dummy := Adjacency_Fly (R, ((O, X, X),
                                        (X, X, X),
                                        (O, X, X)), (2, 2));
     ELSIF Organization = Hex_Odd
       THEN Dummy := Adjacency_Fly (R, ((X, X, O),
                                        (X, X, X),
                                        (X, X, O)), (2, 2));
     ELSIF A = c4
       THEN Dummy := Adjacency_Fly (R, ((O, X, O),
                                        (X, X, X),
                                        (O, X, O)), (2, 2));
     ELSIF A = c8
       THEN Dummy := Adjacency_Fly (R, ((X, X, X),
                                        (X, X, X),
                                        (X, X, X)), (2, 2));
       ELSE RAISE Inconsistent_Adjacency;
       END IF;
      Num_Times := Num_Times - 1;
      END LOOP;
    RETURN Dummy;
    END Expand;
```

FIGURE 11: Ada-IPL Expand Algorithm.

for the given adjacency and repeats this the specified number of times. Defining "X" as True and "O" as False simplifies the visualization of the masks.

Shrink

Conceptually the shrink operation is the opposite of the expand operation, even though they do not commute or annihilate each other. Still Shrink is easily defined as expanding the complement of the given region and then taking the complement of this result (Rosenfeld and Kak, 1976, p. 362). Figure 12 is the algorithm which performs this operation.

```
FUNCTION Shrink (R : Region;
                 A : Adjacencies;
                 Num_Times : Integer := 1);
          RETURN Region is
  BEGIN
   RETURN Complement (Expand (Complement(R), A, Num_Times));
  END Shrink;
```

FIGURE 12: Ada-IPL Shrink Algorithm.

Area

The area operation presented here is merely the number of Pats which are in the given region. It is shown in Figure 13. The algorithm demonstrates how the "For Each" control structure is implemented in Ada using the Next_Member function. The Next_Member function returns the coordinates of Pats in the region in some unspecified order.

In order for the function to know where in the "list of members" it is located, it is given a "previous" Pat's coordinates. If it is given null coordinates then it returns a "first" member. After a "last" member it returns the null coordinates. The order of the members in the "list" as well as which members are "first" or "last" is undefined. The implementation of the "For Each" structure using this Next_Member function is straightforward.

```
FUNCTION Area (R : Region) RETURN Integer is
    Temp : Integer := 0;
    Point : Coordinates := (Nullity => Nil);
  BEGIN
    Point := Next_Member (Point, R);
    WHILE Point.Nullity /= Nil LOOP
      Temp := Temp + 1;
      Point := Next_Member (Point, R);
      END LOOP;
    RETURN Temp;
    END Area;
```

FIGURE 13: Ada-IPL Area Algorithm.

Although it would be much clearer to express this algorithm using a "For Each" loop, it is not possible to add control structures to existing languages without modifying the compiler or requiring a precompiler. When imbedding IPL within a high level language the cost of precompiling or modifying the compiler is generally not worth this increase in clarity. If a compiler were implemented for the IPL language, however, it would certainly be desirable to include a direct implementation of a "For Each" loop rather than rely on equivalent constructs.

CHAPTER SIX
CONCLUSION

An image processing language (IPL) is developed from an examination of the domains and operations which are commonly used to express image processing operations in a strongly typed, high-level language. The motivation behind this approach is to develop a language which is portable and readable. The domains, operations, and a control structure are described for the language. In addition, the language is implemented in Ada and several image processing algorithms are expressed in this implementation.

Summary

This description of IPL assumes the existence of D-type control structures and data type definition capabilities. In addition, IPL defines picture, mask, region, boundary, and histogram data types. The domains for these types are described in terms of a basic element called Pat (for picture atom) whose domain definition is undefined. Thus, each of the above domains is available for any Pat domain. The structures for objects from the picture and mask domains are identical, a two-dimensional array, but their semantic uses in IPL are different. In general, pictures are the results of IPL operations while masks help define the

operations. The structures for objects from the region and boundary domains are not specified. Histogram objects are equivalent to one dimensional arrays which are indexed by either the Pat domain or another domain into which the Pat domain is mapped.

Several operations are defined for these domains. Picture, mask, and region domains have three basic operations; unary, binary, and fly. Unary and binary operations can be completed without reference to the values of neighboring Pats while fly operations require such information. Rotate_90 is also defined for the picture domain. The region domain has the set operations of union, intersection, and division; object creation operations; and component selection operations. The boundary domain has the creation, selection, and set operations listed for regions. It also has a border following operation for traversing the boundary. One object can be registered to another object in IPL by defining a registration, a functional mapping from the coordinates domain of the first object to the coordinates domain of the second. This mapping can be described with a Set_Of_Coordinate_Pairs or as some rule. The set operations (union, intersection, and division), a creation operation, and component selection operations are available for Set_Of_Coordinate_Pairs objects.

One additional control structure, "For Each" is defined for IPL to provide the capability to clearly express the execution of a sequence of operations for each member of a set. This control structure is not implemented directly in the Ada implementation of IPL but is implemented indirectly through a D-type loop structure and a "Next" function. The "Next" function provides a capability to step through all the elements which are members of a set. This differs from the equivalent D-type structure which steps through all potential members of the set and conditions execution of the sequence of operations on set membership. The inclusion of the "For Each" construct improves the readability of IPL code.

Results

The IPL operations form a basic set of operations which can be used to express common image processing algorithms. This satisfies the first goal for the IPL development.

The Ada implementaion of IPL (Ada-IPL) satisfies the second goal, to describe an implementation in an existing, portable, high-level language. Ada was chosen for this implementation because one of its design goals was to develop a portable language. Ada is also a high-level language with the D-type control structures and data type definiton facilities required by IPL. In addition, Ada provides the capability to describe generic subprograms, or templates, which allowed IPL to be implemented without

specifying a Pat domain. This generic capability was also used to define the unary, binary, and fly operations.

Ada effectively separates the actual implementation of operations from their use. Thus, portions of this implementation of IPL may be changed without affecting the correctness of algorithms written it. This satisfies the fourth goal of the IPL development.

The Ada-IPL is not complete nor has it been compiled or tested. Picture input and output operations are not defined since they depend on the environment in which they are used. Also, no Ada compiler was available to check or test this implementation. The Ada-IPL is presented primarily as a demonstration of an implementation of IPL. It also provides a second description of IPL as well as a syntax for expressing algorithms in IPL.

In order to demonstrate IPL's capability, several common image processing algorithms are expressed in Ada-IPL. The picture algorithms are thresholding, Laplacian filtering, differentiation, texture edge detection, template matching by cross correlation, linear and nonlinear filtering, and Sobel edge enhancement. The region algorithms demonstrated are expand (propogate), shrink, and the area of a region. These algorithms were selected based on their common use. Thus, they should be familiar to most readers. In addition, the algorithms demonstrate the use of the IPL unary, binary, and fly operations and they highlight the effect IPL has on their expression.

Further Work

Several areas of further work on IPL are indicated. Obviously, compiling and testing the Ada-IPL is necessary before it can be used as a programming language. Until this is accomplished, it is only useful as a design or specification language. Concurrent with this testing, the implementation of the operations should be modified to improve their performance. The implementations provided were used based on the ease of establishing their correctness and communicating the semantics of the operations. In a testing environment, these operations should be implemented with increased performance where possible.

Additional work is also needed to examine other image processing algorithms to determine if any additional operations are required to describe them. The given operations appear to be adequate to express any image processing task, but their completeness is not proven. Further efforts to find additional operations or demonstrate the completeness of the given operations is required.

Finally, an architecture could be designed to directly implement the unary, binary, and fly operations. The fly operation appears to be the most time intensive of all the IPL operations. By focusing an architecture on improving the performance of the fly as well as the unary and binary operations, a very efficient architecture for image processing can be developed. The nature of these three

operations suggest an architecture similar to that described for the CLIP4 (Duff, 1979), but since the development of this language was purposefully divorced from architectural considerations, the subject of an architecture for the language requires much closer examination.

REFERENCES

Ada Programming Language, ANSI/MIL-STD-1815A, 22 January 1983, Naval Publications and Forms Center, Phildelphia, Pa. (1983).

Bulman, David M., "Is Ada the Answer?" The Yourdon Report, vol. 6-6, 7-1, Yourdon Inc., New York (1981).

Chang, Nina-San, Image Analysis and Image Data Base Management, UMI Research Press, Ann Arbor, Michigan (1981).

Douglas, R. J., "MAC: A Programming Language for Asynchronous Image Processing," in Languages and Architectures for Image Processing, Duff M. J. B., and Levialdi, S. (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 41-52 (1981).

Duff, M. J. B., "Parallel Processors for Digital Image Processing," in Advances in Digital Image Processing, Stucki, P. (ed.), Plenum Publishing Corp., New York, pp. 265-276 (1979).

Duff, M. J. B., and Levialdi, S., Languages and Architectures for Image Processing, Harcourt Brace Jovanovich, Publishers, New York (1981).

Foley, James D., and Van Dam, Andries, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts (1983).

Gudmundsson, B., "Overview of the High-level Language for PICAP," in Languages and Architectures for Image Processing, Duff, M. J. B., and Levialdi, S. (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 147-156 (1981).

Kulpa, Z., "PICASSO, PICASSO-SHOW, and PAL -- A Development of a High-level Software System for Image Processing," in Languages and Architectures for Image Processing, Duff, M. J. B., and Levialdi, S. (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 13-24 (1981).

Ledgard, Henry and Marcotty, Michael, The Programming Language Landscape, Science Research Associates, Inc., Chicago, Ill. (1981).

Levialdi, S., Maggiolo-Schettini, A., Napoli, M., Tortora, G., and Uccella, G., "On the Design and Implementation of PIXAL, a Language for Image Processing," in Languages and Architectures for Image Processing, Duff, M. J. B., and Levialdi, S., (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 89-98 (1981).

Maggiolo-Schettini, A., "Comparing Some High-level Languages for Image Processing," in Languages and Architectures for Image Processing, Duff, M. J. B., and Levialdi, S. (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 157-164 (1981).

Pratt, William K., Digital Image Processing, John Wiley and Sons, Inc., New York (1978).

Radhakrishnan, T., Barrera, R., Guzman, A., and Jinich, A., "Design of a High-level Language (L) for Image Processing in Languages and Architectures for Image Processing, Duff, M. J. B., and Levialdi, S. (ed.), Harcourt Brace Jovanovich, Publishers, New York, pp. 25-40 (1981).

Rosenfeld, Azriel and Kak, Avinash C., Digital Picture Processing, Harcourt Brace Jovanovich, Publishers, New York (1976).

Sinha, R. M. K., "PLANG -- A Picture Language Schema for a Class of Pictures," Pattern Recognition, vol. 16, no. 4, pp. 373-383 (1983).

Williams, Robin, "Image Processing and Computer Graphics," in Advances in Digital Image Processing, Stucki, P. (ed.), Plenum Publishing Corp., New York, pp. 219-234 (1979).

APPENDIX A
ADA IMPLEMENTATION OF IPL

```
GENERIC
    TYPE Pats is PRIVATE;
    Organization : IN (Orthogonal, Hex_even, Hex_odd)
                        := Orthogonal;
    Mask_Index_First : IN Integer := 1;

  PACKAGE IPL is
    TYPE Index is Integer;
    TYPE Pictures is
             ARRAY (Index RANGE <>, Index RANGE <>) OF Pats;
    Incompatible_Pictures : EXCEPTION;

--Picture Operations
    GENERIC
        TYPE Element_1 is PRIVATE;
        TYPE Picture_1 is ARRAY
               (Index RANGE <>, Index RANGE <>) OF Element_1;
        TYPE Element_2 is PRIVATE;
        TYPE Picture_2 is ARRAY
               (Index RANGE <>, Index RANGE <>) OF Element_2;
        TYPE Element_3 is PRIVATE;
        TYPE Picture_3 is ARRAY
               (Index RANGE <>, Index RANGE <>) OF Element_3;
        WITH F(Left : Element_1; Right : Element_2)
                    RETURN Element_3 is <>;
      FUNCTION Picture_Binary_Op
                    (Left : Picture_1; Right : Picture_2)
           RETURN Picture_3;
      -- Left and Right must be of the same size
      -- i.e.  Left'LENGTH = Right'LENGTH
      -- Incompatible_Pictures is raised if they are not
```

```
GENERIC
    TYPE Element_1 is PRIVATE;
    TYPE Picture_1 is ARRAY
          (Index RANGE <>, Index RANGE <>) OF Element_1;
    TYPE Element_3 is PRIVATE;
    TYPE Picture_3 is ARRAY
          (Index RANGE <>, Index RANGE <>) OF Element_3;
    WITH F(Left : Element_1) RETURN Element_3 is <>;
  FUNCTION Picture_Unary_Op (Left : Picture_1)
       RETURN Picture_3;

FUNCTION Rotate_90 (P : Pictures; Times : Natural)
       RETURN Pictures;

PROCEDURE Input_Picture (P : OUT Pictures);
PROCEDURE Output_Picture (P : IN Pictures);

TYPE Nullities is (Nil, Valid);
TYPE Coordinates is RECORD (Nullity : Nullities := Nil;
                            Row_First : Index;
                            Row_Last : Index;
                            Column_First : Index;
                            Column_Last : Index)
     CASE Nullity is
       WHEN Nil => NULL;
       WHEN Valid =>
         Row : Index RANGE Row_First..Row_last;
         Column : Index RANGE Column_First..Column_Last;
     END RECORD;
SUBTYPE Valid_Coordinates is
                Coordinates (Nullity => Valid);

TYPE Histograms is ARRAY (Pats RANGE <>) of Natural;
```

```
-- Mask Operations
  SUBTYPE Mask_Index is
                Index RANGE Mask_Index_First..Max_Int;
  TYPE Pat_Mask is ARRAY
       (Mask_Index RANGE <>, Mask_Index RANGE <>) OF Pats;
-- Mask objects can be explicitly converted to type Picture

  GENERIC
      TYPE Mask_Index is RANGE <>;  -- Integer Type
      TYPE Element_1 is PRIVATE;
      TYPE Mask_1 is ARRAY
              (Mask_Index RANGE <>, Mask_index RANGE <>)
            OF Element_1;
      TYPE Element_2 is PRIVATE;
      TYPE Mask_2 is ARRAY
              (Mask_Index RANGE <>, Mask_index RANGE <>)
            OF Element_2;
      TYPE Picture_2 is ARRAY
            (Index RANGE <>, Index RANGE <>) of Element_2;
      TYPE Element_3 is PRIVATE;
      TYPE Picture_3 is ARRAY
            (Index RANGE <>, Index RANGE <>) of Element_3;
      WITH FUNCTION Border_Value (I, J : Index)
                RETURN Pats is Zero_Border;
      WITH F (M1 : Mask_1; M2 : Mask_2)
                RETURN Element_3 is <>;
   -- F is called by Fly only with masks of the same size
   -- i. e., M1'Length(1) = M2'Length(1)
   --    AND M1'Length(2) = M2'Length(2)
    FUNCTION Fly (P2 : Picture_2;
                  M1 : Mask_1;
                  Mask_Center : Valid_Coordinates)
         RETURN Picture_3;
       -- P2 and P3 must be the same size
       --  i.e. P2'LENGTH = P3'LENGTH
       -- Incompatible_Pictures is raised if they are not
  GENERIC
      TYPE Element_1 is PRIVATE;
      TYPE Mask_1 is ARRAY
              (Mask_Index RANGE <>, Mask_index RANGE <>)
            OF Element_1;
      TYPE Element_2 is PRIVATE;
      TYPE Mask_2 is ARRAY
              (Mask_Index RANGE <>, Mask_index RANGE <>)
            OF Element_2;
      TYPE Element_3 is PRIVATE;
      WITH "*" (E1 : Element_1; E2 : Element_2)
               RETURN Element_3) is <>;
    FUNCTION Linear_Combination (M1 : Mask_1; M2 : Mask_2)
         RETURN Element_3;
    -- M1 and M2 must be of the same size
    -- or Constraint error is raised
```

```
    TYPE Adjacencies is (C4, C8, C6);
    FUNCTION Adjacent (C1, C2 : Valid_Coordinates;
                       A : Adjacencies) RETURN Boolean;
    Inconsistent_Adjacencies : EXCEPTION;

-- Region Operations
    TYPE Region (Row_First : Index;
                 Row_Last : Index;
                 Column_First : Index;
                 Column_Last : Index) is PRIVATE;
    FUNCTION Pat_Value (R : Region; C : Valid_Coordinates)
         RETURN Pats;
    FUNCTION Make_Pat_Region
       (P : Pictures; C : Valid_Coordinates) RETURN Region;
    TYPE Boolean_Pictures is ARRAY
           (Index RANGE <>, Index RANGE <>) OF Boolean;
    FUNCTION Make_Region
       (P : Pictures; BP : Boolean_Pictures) RETURN Region;
    FUNCTION In_Region (R : Region; C : Valid_Coordinates)
         RETURN Boolean;
    FUNCTION "+" (R1, R2 : Region) RETURN Region;
                 -- Union
    FUNCTION "*" (R1, R2 : Region) RETURN Region;
                 -- Intersection
    FUNCTION "-" (R1, R2 : Region) RETURN Region;
                 -- Difference
    FUNCTION Complement (R : Region) RETURN Region;
    FUNCTION Picture_Of (R : Region) RETURN Pictures;
    FUNCTION Next_Member (C : Coordinates; R : Region)
         RETURN Coordinates;
      -- C must be in R or the Null Coordinates
    Incompatible_Regions : EXCEPTION;

  --Region Binary Operation
    GENERIC
        TYPE Element_2 is PRIVATE;
        TYPE Picture_2 is ARRAY
              (Index RANGE <>, Index RANGE <>) OF Element_2;
        WITH F(Left : Boolean; Right : Element_2)
                    RETURN Boolean is <>;
      FUNCTION Region_Binary_Op
           (Left : Region; Right : Picture_2) RETURN Region;
      -- Left and Right must be of the same size
      -- i.e.  Left.Last - Left.First = Right'LENGTH
      -- Incompatible_Regions will be raised if they are not

    -- Region Mask Operations
    TYPE Boolean_Mask is ARRAY
           (Mask_Index RANGE <>, Mask_Index RANGE <>)
           of Boolean;
```

```
    GENERIC
        TYPE Mask_Index is RANGE <>;  -- Integer Type
        TYPE Element_1 is PRIVATE;
        TYPE Mask_1 is ARRAY
                (Mask_Index RANGE <>, Mask_index RANGE <>)
               OF Element_1;
        WITH FUNCTION Border_Value (I, J : Index)
                    RETURN Boolean is Not_In_Border;
        WITH F (M1 : Mask_1; M2 : Boolean_Mask)
                   RETURN Boolean is <>;
      -- F is called by Fly only with masks of the same size
      -- i.e. M1'Length(1) = M2'Length(1)
      --  AND M1'Length(2) = M2'Length(2)
      FUNCTION Region_Fly (R : Region;
                           M1 : Mask_1;
                           Mask_Center : Valid_Coordinates)
            RETURN Region;


--Boundary Operations
    TYPE Boundary (Row_First : Index;
                   Row_Last : Index;
                   Column_First : Index;
                   Column_Last : Index) is PRIVATE;
    TYPE Boundary_Classifications is (On, Off, Limit);
    FUNCTION Pat_Value (B : Boundary; C : Valid_Coordinates)
          RETURN Pats;
    FUNCTION In_Region (B : Boundary; C : Valid_Coordinates)
          RETURN Boolean;
    FUNCTION Make_Boundary (R : Region; A : Adjacency)
         RETURN Boundary;
    FUNCTION On_Boundary (B : Boundary;
                          C : Valid_Coordinates)
         RETURN Boundary_Classification;
    TYPE Direction is (Clockwise, Counterclockwise);
    FUNCTION Next_Pat (B : Boundary;
                       A : Adjacency;
                       D : Direction;
                       C_On, C_previous : Coordinates)
         RETURN Coordinates;
    FUNCTION Region_Of (B : Boundary) RETURN Region;
    Incompatible_Coordinates : EXCEPTION;
```

```
-- Registration
  GENERIC
     From_Picture : IN Pictures;
     To_Picture : IN Pictures;
   PACKAGE Coordinate_Pair_Sets is
     TYPE Coordinate_Pair_Set is PRIVATE;
     TYPE From_Coordinates is Valid_Coordinates
                (Row_First => From_Picture'FIRST(1),
                 Row_Last => From_Picture'LAST(1),
                 Column_First => From_Picture'FIRST(2),
                 Column_Last => From_Picture'LAST(2));
     TYPE To_Coordinates is Coordinates
                (Row_First => To_Picture'FIRST(1),
                 Row_Last => To_Picture'LAST(1),
                 Column_First => To_Picture'FIRST(2),
                 Column_Last => To_Picture'LAST(2));
     TYPE Coordinate_Pairs is
              RECORD (Nullity : Nullities := Nil)
         CASE Nullity is
           WHEN Nil => NULL;
           WHEN Valid =>
              From : From_Coordinates;
              To : To_Coordinates;
         END Record;
     SUBTYPE Valid_Coordinate_Pairs is
                  Coordinate_Pairs (Nullity => Valid);
     Null_CPS : CONSTANT Coordinate_Pair_Set;
     Inconsistent_CPS : EXCEPTION;
-- This implementation does not use this error.
-- Its occurance is prevented by the package parameters
-- restricting calls to these functions with CPSs from
-- the input pictures only (From and To)
     Inconsistent_Registration : EXCEPTION;
     FUNCTION "+" (Set_1, Set_2 : Coordinate_Pair_Set)
          RETURN Coordinate_Pair_Set;    -- UNION
     FUNCTION "*" (Set_1, Set_2 : Coordinate_Pair_Set)
          RETURN Coordinate_Pair_Set;    -- INTERSECTION
     FUNCTION "-" (Set_1, Set_2 : Coordinate_Pair_Set)
          RETURN Coordinate_Pair_Set;    -- DIFFERENCE
     FUNCTION Convert_To_Set (CP : Valid_Coordinate_Pairs)
          RETURN Coordinate_Pair_Set;
     FUNCTION Next_Element (CP : Coordinate_Pairs;
                            Set : Coordinate_Pair_Set)
          RETURN Coordinate_Pairs;
    -- CP must be in Set or the Null_Registerd_Coordinates
     FUNCTION Registered_Coordinates
        (C1 : From_Coordinates; Set : Coordinate_Pair_Set)
          RETURN To_Coordinates;
     FUNCTION Change_Registered_Coordinates
                   (CP : Valid_Coordinate_Pairs;
                    Set : Coordinate_Pair_Set)
          RETURN Coordinate_Pair_Set;
```

```
      GENERIC
         WITH FUNCTION Map (From_pt: From_Coordinates;
                            Enumerated_Map: Coordinate_Pair_Set)
                     RETURN To_Coordinates is <>;
         From_Picture: IN Pictures;
         To_Picture: IN Pictures;
         Registered_Map : Coordinate_Pair_Set;
       PACKAGE Register is
         No_Registered_Value : EXCEPTION;
         FUNCTION Registered_Coordinates
                           (From_Point: From_Coordinates)
              RETURN To_Coordinates;
         FUNCTION Registered_Value
              (From_Point: From_Coordinates) RETURN Pats;
       END Register;
      PRIVATE
       TYPE Coordinate_Pair_Set is ARRAY
             (From_Picture'RANGE(1), From_Picture'RANGE(2))
          OF To_Coordinates;
       Null_CPS : CONSTANT Coordinate_Pair_Set
                     := (OTHERS => (Nullity => Nil));
      END Coordinate_Pair_Sets;


   PRIVATE
      TYPE Region is RECORD (Row_First : Index;
                             Row_Last : Index;
                             Column_First : Index;
                             Column_Last : Index)
            Base_Picture : Pictures (Row_First..Row_Last,
                                     Column_First..Column_Last);
            Region_Members : ARRAY
                           (Index RANGE Row_First..Row_Last,
                      Index RANGE Column_First..Column_Last)
                      of Boolean;
          END Record
      TYPE Boundary is RECORD (Row_First : Index;
                               Row_Last : Index;
                               Column_First : Index;
                               Column_Last : Index)
            Base_Region : Region (Row_First, Row_Last,
                                  Column_First, Column_Last);
            Boundary_Members : ARRAY
                    (Index RANGE Row_First..Row_Last,
                     Index RANGE Column_First..Column_Last)
                    of Boundary_Classifications
                  := (OTHERS => Off);
            Boundary_Adjacency : Adjacency;
          END Record;
END IPL;
```

APPENDIX B
BODY FOR THE ADA IMPLEMENTATION OF IPL

```
PACKAGE BODY IPL is

      FUNCTION Picture_Binary_Op (Left : Picture_1;
                                  Right : Picture_2)
           RETURN Picture_3 is
          I, J : Index;
          Dummy : Pictures (Left'RANGE(1), Left'RANGE(2));
        BEGIN
          IF NOT ((Left'Range(1) = Right'Range(1))
                  OR (Left'RANGE(2) = Right'RANGE(2)))
            THEN RAISE Incompatible_Pictures;
            ENDIF;
          FOR I in Left'RANGE(1) LOOP
            FOR J in Right'RANGE(2) LOOP
              Dummy (I, J) := F (Left(I, J), Right(I, J));
              END LOOP;
            END LOOP
          RETURN Dummy;
          END Picture_Binary_Op;

      FUNCTION Rotate_90 (P : Pictures; Times : Positive)
           RETURN Pictures is
          I, J, N : Index;
          Dummy : Pictures (P'RANGE(1), P'RANGE(2));
        BEGIN
          WHILE Times /= 0 LOOP
            N := Dummy'LAST(2);
            FOR I in P'RANGE(1) LOOP
              FOR J in P'RANGE(2) LOOP
                Dummy (J, N) := P (I, J);
                END LOOP;
              N := Index'PRED(N);
              END LOOP;
            Times := Times - 1;
            END LOOP;
          RETURN Dummy;
          END Rotate_180;
```

```
FUNCTION Picture_Unary_Op (Left : Picture_1)
     RETURN Picture_3 is
    I, J : Index;
    Dummy : Pictures(Left'RANGE(1), Left'RANGE(2));
  BEGIN
    FOR I in Left'RANGE(1) LOOP
      FOR J in Right'RANGE(2) LOOP
        Dummy (I, J) := F (Left(I, J));
        END LOOP;
      END LOOP
    RETURN Dummy;
    END Picture_Unary_Op;

FUNCTION Adjacent (C1, C2 : Valid_Coordinates)
     RETURN Boolean is
    Adj : Boolean;
  BEGIN
    IF (((A = C6) AND (Organization = Orthogonal))
      OR ((A /= C6) AND (Organization /= Orthogonal)))
      THEN RAISE Adjacency_Error;
      ENDIF;
    Adj := ((((Abs (C1.row - C2.row)) = 1)
             AND (C1.Column = C2.Column))
            OR (((Abs (C1.Column - C2.Column)) = 1)
                AND (C1.Row = C2.Row)));
    IF NOT Adj
      THEN IF A = .8
         THEN Adj := (((Abs (C1.Row - C2.Row)) = 1)
             AND ((Abs (C1.Column - C2.Column)) = 1));
         ELSEIF ((A = C6)
                 AND ((Abs (C1.Row - C2.Row)) = 1)
           THEN IF C2.Column - C1.Column = 1
              THEN Adj := (((Organization = Hex_Even)
                        AND ((Rem (C1.Row, 2)) = 1))
                        OR ((Organization = Hex_Odd)
                        AND ((Rem (C1.Row, 2)) = 0)));
              ELSEIF C1.Column - C2.Column = 1
               THEN Adj := (((Organization = Hex_Even)
                        AND (Rem (C1.Row, 2)) = 0))
                        OR ((Organization = Hex_Odd)
                        AND ((Rem (C1.Row, 2)) = 1)));
              ENDIF;
         ENDIF;
      ENDIF;
    RETURN Adj;
    END Adjacent;

FUNCTION Zero_Border (I, J: Index; P: Pictures)
     RETURN Pats is
BEGIN
  RETURN 0;
END Zero_Border;
```

```
FUNCTION Fly (P2 : Picture_2;
              M1 : Mask_1;
              Mask_center : IN Valid_Coordinates)
     RETURN Pictures is
    Dummy : Pictures (P2'RANGE(1), P2'RANGE(2));
    M2 : Mask_2 (M1'RANGE(1), M1'RANGE(2));
    I, J, I_Offset, J_Offset, K, N : Index;
    Hex_Correction : Index RANGE 1..0;
  BEGIN
     FOR I IN P2'RANGE(1) LOOP
       FOR J IN P2'RANGE(2) LOOP
         IF (Organization /= Orthogonal)
            AND THEN ((Rem (Mask_Center.Row, 2))
                        = (Rem (I, 2)))
           THEN Hex_Correction := 0;
           ELSE Hex_Correction := 1;
           ENDIF;
         FOR K IN M2'RANGE(1) LOOP
           FOR N IN M2'RANGE(2) LOOP
             I_Offset := I + Mask_Center.Row - K;
             J_Offset := J + Mask_Center.Column - N
                           + Hex_Correction;
             IF (I_Offset IN P2'RANGE(1))
                 AND (J_Offset IN P2'RANGE(2))
               THEN M2(K,N) := P2(I_Offset, J_Offset);
               ELSE M2(K,N) :=
                     Border_Value(I_Offset, J_Offset);
               ENDIF;
             END LOOP;
           END LOOP;
         Dummy(I,J) := F(M1, M2);
         END LOOP;
       END LOOP;
     RETURN Dummy;
     END Fly;

FUNCTION Linear_Combination (M1 : Mask_1; M2 : Mask_2)
     RETURN Element_3 is
-- M1 and M2 must be of the same size
-- or Constraint_Error is raised
    IM1, JM1, IM2, JM2 : Index;
    TEMP : Element_3 := 0;
  BEGIN
    IM1 := M1'FIRST(1);     JM1 := M1'FIRST(2);
    FOR IM2 in M2'RANGE(1) LOOP
      FOR JM2 in M2'RANGE(2) LOOP
        Temp := Temp + (M1(IM1, JM1) * M2(IM2, JM2));
        IM1 := Index'SUCC(IM1);
        END LOOP;
      JM1 := Index'SUCC(JM1);
      END LOOP;
    RETURN Temp;
    END Linear_Combination;
```

```
PACKAGE BODY Coordinate_Pair_Sets is
   FUNCTION "+" (Set_1, Set_2 : Coordinate_Pair_Set)
        RETURN Coordinate_Pair_Set is   -- UNION
     I, J : Index;
     Dummy . Coordinate_Pair_Set
                   (Set_1'RANGE(1), Set_2'RANGE(2))
                   := (OTHERS => (Nullity => Nil));
   BEGIN
     FOR I IN Set_1'RANGE(1) LOOP
       FOR J IN Set_1'RANGE(2) LOOP
         IF Set_1(I, J) /= (Nullity => Nil)
            THEN IF Set_2(I, J) = (Nullity => Nil)
                   OR Set_1(I, J) = Set_2(I, J)
                 THEN Dummy(I, J) := Set_1(I, J);
                 ELSE RAISE Inconsistent_Registration;
                 ENDIF;
            ELSE Dummy(I, J) := Set_2(I, J);
            ENDIF;
            ENDLOOP;
          ENDLOOP;
     RETURN Dummy;
     END "+";

   FUNCTION "*" (Set_1, Set_2 : Coordinate_Pair_Set)
        RETURN Coordinate_Pair_Set is  -- INTERSECTION
     I, J : Index;
     Dummy : Coordinate_Pair_Set
              (Set_1'RANGE(1), Set_2'RANGE(2))
              := (OTHERS => (Nullity => Nil));
   BEGIN
     FOR I IN Set_1'RANGE(1) LOOP
       FOR J IN Set_2'RANGE(2) LOOP
         IF Set_1(I, J) /= (Nullity => Nil)
            THEN IF Set_1(I, J) = Set_2(I, J)
                 THEN Dummy(I, J) := Set_1(I, j);
                 ELSEIF Set_2(I, J) /= (Nullity => Nil)
                   THEN RAISE Inconsistent_Registration;
                 ENDIF;
            ENDIF;
            ENDLOOP;
          ENDLOOP;
     RETURN Dummy;
     END "*";

   FUNCTION Convert_To_Set (CP : Valid_Coordinate_Pairs)
        RETURN Coordinate_Pair_Set is
     BEGIN
       RETURN (CP.From_Coordinates.Row =>
                   (CP.From_Coordinates.Column =>
                       CP.To_Coordinates),
               OTHERS => (OTHERS => (Nullity => Nil)));
     END Convert_To_Set;
```

```
FUNCTION "-" (Set_1, Set_2 : Coordinate_Pair_Set)
     RETURN Coordinate_Pair_Set is   -- DIFFERENCE
    I, J : Index;
    Dummy : Coordinate_Pair_Set
              (Set_1'RANGE(1), Set_2'RANGE(2))
              := (OTHERS => (Nullity => Nil));
  BEGIN
    FOR I IN Set_1'RANGE(1) LOOP
      FOR J IN Set_2'RANGE(2) LOOP
        IF Set_1(I, J) /= (Nullity => Nil)
          THEN IF Set_2(I, J) = (Nullity => Nil)
               THEN Dummy(I, J) := Set_1(I, J);
               ELSEIF Set_1(I, J) /= Set_2(I, J)
                THEN RAISE Inconsistent_Registration;
               ENDIF;
          ENDIF;
        ENDLOOP;
      ENDLOOP;
    RETURN Dummy;
    END "-";

FUNCTION Next_Element (CP : Coordinate_Pairs;
                       Set : Coordinate_Pair_Set)
     RETURN Coordinate_Pairs is
-- CP must be in Set or the Null_Registerd_Coordintes
    I, J : Index;
  BEGIN
    IF CP.Nullity = Nil
      THEN I := Set'First(1);
           J := Set'First(2);
           IF Set(I, J).Nullity /= Nil
             THEN RETURN (Valid,
                          From => (I, J),
                          To => (Set(I, J));
           ENDIF;
      ELSE I := CP.From.Row;
           J := CP.From.Column;
      ENDIF;
    WHILE (I /= Set'Last(1))
          AND (J /= Set'Last(2)) LOOP
      IF J = Set'Last(2)
        THEN J := Set'First(2);
             I := Set'RANGE(1)'Succ(I);
        ELSE J:= Set'RANGE(2)'Succ(J);
        ENDIF;
      IF Set(I, J).Nullity /= Nil
        THEN RETURN (Valid, (I, J), Set(I, J));
        ENDIF;
      ENDLOOP;
    RETURN (Nullity => Nil);
  END Next_Element;
```

```
FUNCTION Registered_Coordinates
            (Cl : From_Coordinates;
             Set : Coordinate_Pair_Set)
      RETURN To_Coordinates is
   BEGIN
     RETURN Set (Cl.Row, Cl.Column);
   END Registered_Coordinates;

FUNCTION Change_Registered_Coordinates
             (CP : Valid_Coordinate_Pairs;
              Set : Coordinate_Pair_Set)
      RETURN Coordinate_Pair_Set is
  BEGIN
    Set(CP.From_Coordinates.Row,
        CP.From_Coordinates.Column)
      := CP.To_Coordinates;
    Return Set;
  END Change_Registered_Coordinates;


 PACKAGE BODY Register is
   FUNCTION Registered_Coordinates
                  (From_Point: From_Coordinates)
       RETURN To_Coordinates is
      BEGIN
        RETURN Map (From_Point, Registered_Map);
      END Registered_Coordinates;

   FUNCTION Registered_Value
                  (From_Point: From_Coordinates)
       RETURN Pats is
      Pt : To_Coordinatesl
     BEGIN
       Pt := Map (From_Point, Registered_Map);
       IF Pt.Nullity = Nil
         THEN RAISE No_Registered_Value;
         ENDIF;
       RETURN To_Picture(Pt.Row, Pt.Column);
     END Registered_Value;

  END Register;
END Coordinate_Pair_Sets;
```

```
-- Region Operations
    FUNCTION Pat_Value (R : Region; C : Valid_Coordinates)
         RETURN Pats is
      BEGIN
        RETURN R.Base_Picture (C.Row, C.Column);
      END Pat_Value;

    FUNCTION Make_Pat_Region (P : Pictures;
                              C : Valid_Coordinates)
         RETURN Region is
        Dummy : Region (P'FIRST(1), P'LAST(1),
                        P'FIRST(2), P'LAST(2));
      BEGIN
        Dummy.Base_Picture := P;
        Dummy.Region_Members (C.Row, C.Column) := True;
        RETURN Dummy;
      END Make_Pat_Region;

    FUNCTION Make_Region (P : Pictures;
                          BP : Boolean_Pictures)
         RETURN Region is
        Dummy : Region (P'FIRST(1), P'LAST(1),
                        P'FIRST(2), P'LAST(2));
      BEGIN
        IF (P'LENGTH(1)) /= BP'LENGTH(1)
            OR (P'LENGTH(2) /= BP'LENGTH(2))
          THEN RAISE Incompatible_Pictures;
          END IF;
        Dummy.Base_Picture := P;
        Dummy.Region_Members := BP;
        RETURN Dummy;
      END Make_Region;

    FUNCTION In_Region (R : Region; C : Valid_Coordinates)
         RETURN Boolean is
      BEGIN
        RETURN R.Region_Members (C.Row, C.Column);
      END In_Region;

    FUNCTION "+" (R1, R2 : Region) RETURN Region is -- Union
        I : Index;
      BEGIN
        IF R1.Base_Picture /= R2.Base_Picture
          THEN RAISE Incompatible_Regions;
          ENDIF;
        FOR I IN RANGE R1.Row_First..R1.Row..Last LOOP
          R1.Region_Members(I) := R1.Region_Members(I)
                                  OR R2.Region_Members(I);
        ENDLOOP;
        RETURN R1;
        END "+";
```

```
FUNCTION "*" (R1, R2 : Region) RETURN Region is
             -- Intersection
    I : Index;
  BEGIN
    IF R1.Base_Picture /= R2.Base_Picture
      THEN RAISE Incompatible_Regions;
      ENDIF;
    FOR I IN RANGE R1.Row_First..R1.Row_Last LOOP
      R1.Region_Members(I) := R1.Region_Members(I)
                              AND R2.Region_Members(I);
      ENDLOOP;
    RETURN R1;
    END "*";

FUNCTION "-" (R1, R2 : Region) RETURN Region is
             -- Difference
    I : Index;
  BEGIN
    IF R1.Base_Picture /= R2.Base_Picture
      THEN RAISE Incompatible_Regions;
      ENDIF;
    FOR I IN RANGE R1.Row_first..R1,Row_Last LOOP
      R1.Region_Members(I) := R1.Region_Members(I) AND
                             (NOT R2.Region_Members(I));
      ENDLOOP;
    RETURN R1;
    END "-";

FUNCTION Complement (R : Region) RETURN Region;
    Dummy : Region (R.Row_First, R.Row_Last,
                    R.Column_First, R.Column_Last);
    I : Index;
  BEGIN
    Dummy.Base_Picture := R.Base_Picture;
    FOR I IN R.Row_First..R.Row_Last LOOP
     Dummy.Region_Members(I) := NOT R.Region_Members(I);
     END LOOP;
    RETURN Dummy;
    END Complement;

FUNCTION Picture_Of (R : Region) RETURN Pictures is
  BEGIN
    RETURN R.Base_Picture;
  END Picture_Of;
```

```
FUNCTION Next_Member (C : Coordinates; R : Region)
     RETURN Coordinates is
 -- C must be in R or the Null Coordinates
       I, J : Index;
     BEGIN
       IF C.Nullity = Nil
         THEN I := R.Row_First;
              J := R.Column_First;
              IF In-Region (R, (I, J))
                THEN RETURN (Valid, Row => I,
                             Column => J);
                ENDIF;
         ELSE I := C.Row;
              J := C.Column;
         ENDIF;
        WHILE (I /= R.Row_Last)
              AND (J /= R.Column_Last) LOOP
         IF J = R.Column_Last
           THEN J := R.Column_First;
                I := Index'Succ(I);
           ELSE J:= Index'Succ(J);
           ENDIF;
         IF In_Region (R, (I, J))
           THEN RETURN (Valid, I, J);
           ENDIF;
         ENDLOOP;
        RETURN (Nullity => Nil);
     END Next_Member;

  FUNCTION Region_Binary_Op (Left : Region;
                             Right : Picture_2)
       RETURN Region;
  -- Left and Right must be of the same size
  -- i.e. Left.Last - Left.First = Right'LENGTH
  -- Incompatible_Regions is raised if they are not
      I, J : Index;
    Dummy : Region (R.Row_First, R.Row_Last,
                    R.Column_First, R.Column_Last);
    BEGIN
      IF NOT ((R.Row_Last-R.Row_First = Right'Range(1))
              OR (R.Column_Last - R.Column_First
                                = Right'RANGE(2)))
        THEN RAISE Incompatible_Regions;
        ENDIF;
      Dummy.Base_Picture := Left.Base_Picture;
      FOR I in Right'RANGE(1) LOOP
        FOR J in Right'RANGE(2) LOOP
          Dummy.Region_Members (I, J)
            := F(Left.Region_Members(I,J), Right(I,J));
        END LOOP;
      END LOOP
      RETURN Dummy;
      END Picture_Binary_Op;
```

```
FUNCTION Not_In_Border(I, J : Index) RETURN Boolean is
  BEGIN
    RETURN False;
  END Not_In_Border;

FUNCTION Region_Fly (R : Region;
                     M1 : Mask_1;
                     Mask-Center : Valid_Coordinates)
     RETURN Region;
    Dummy : Region (R.Row_First, R.Row_Last,
                    R.Column_First, R.Column_Last);
    M2 : Boolean_Mask (M1'RANGE(1), M1'RANGE(2));
    I, J, I_Offset, J_Offset, K, N : Index;
    Hex_Correction : Index RANGE 1..0;
  BEGIN
     FOR I IN R.Row_First..R.Row_Last LOOP
       FOR J IN R.Column_First..R.Column_Last LOOP
         IF (Organization /= Orthogonal)
            AND THEN ((Rem (Mask_Center.Row, 2))
                          = (Rem (I, 2)))
           THEN Hex_Correction := 0;
           ELSE Hex_Correction := 1;
           ENDIF;
         FOR K IN M2'RANGE(1) LOOP
           FOR N IN M2'RANGE(2) LOOP
             I_Offset := I + Mask_Center.Row - K;
             J_Offset := J + Mask_Center.Column
                          - N + Hex_Correction;
             IF (I_Offset IN R.Row_First..R.Row_Last)
                 AND (J_Offset IN
                        R.Column_First..R.Column_Last)
               THEN M2(K,N) :=
                 R.Region_Memebers(I_Offset,J_Offset);
               ELSE M2(K,N) :=
                      Border_Value(I_Offset, J_Offset);
               ENDIF;
             END LOOP;
           END LOOP;
         Dummy.Region_Members(I,J) := F(M1, M2);
         END LOOP;
       END LOOP;
     Dummy.Base_Picture := R.Base_Picture;
     RETURN Dummy;
     END Region_Fly;
```

```
-- Boundary Operations

    FUNCTION Make_Boundary (R : Region; A : Adjacency)
         RETURN Boundary is
        Dummy : Boundary (R.Row_First, R.Row_Last,
                          R.Column_First, R.Column_Last);
        C : Coordinates := (Nullity => Nil);
        I, J, K, N : Index;
        Done : Boolean;
      BEGIN
        Dummy.Base_Region := R;
        Dummy.Boundary_Adjacency := A;
        C := Next_Member (C, R);
        WHILE C.Nulllity /= Nil LOOP
          Done := False;
          K := 0;
          N := -1;
          WHILE NOT Done LOOP
            CASE K is
              WHEN -1 => N := N + 1;
              WHEN  0 => K := K + N;
              WHEN +1 => N := N - 1;
              WHEN OTHERS => NULL;
              END CASE;
            IF Abs (N) = 2
              THEN N := N/2;
              K := 0;
              ENDIF;
            I := C.Row + K;
            J := C.Column + N;
            IF (I IN RANGE R.Row_First..R.Row_Last)
               AND(J IN RANGE R.Column_First..R.Column_Last)
               AND THEN (Adjacent (C, (I, J), A))
               AND THEN (NOT R.Region_Members (I, J))
              THEN Dummy.Boundary_Members (I, J) := On;
              Done := True;
            ELSEIF (K = 0) AND (N = -1)
               THEN Done := True;
                IF (Dummy.Boundary_Members
                                    (C.Row, C.Column) = Off)
                    AND THEN ((C.Row = R.Row_First) OR ELSE
                              (D.Row = R.Row_Last) OR ELSE
                              (C.Column = R.Column_First)
                            OR ELSE (C.Column=R.Column_Last))
                 THEN Dummy.Boundary_Members
                                  (C.Row, C.Column) := Limit;
                 ENDIF;
              ENDIF;
            END LOOP;
          C := NEXT_Member (C, R);
          END LOOP;
        END Make_Boundary;
```

```
FUNCTION Pat_Value (B : Boundary; C : Valid_Coordinates)
     RETURN Pats is
  BEGIN
    RETURN B.Base_Region.Base_Picture (C.Row, C.Column);
  END Pat_Value;

FUNCTION In_Region ( B : Boundary; C : Valid_Coordinates)
     RETURN Boolean is
  BEGIN
    RETURN B.Base_Region.Region_Members(C.Row,C.Column);
  END In_Region;

FUNCTION On_Boundary (B : Boundary;
                      C : Valid_Coordinates)
     RETURN Boundary_Classification is
  BEGIN
    RETURN B.Boundary_Members (C.Row, C.Column);
  END On_Boundary;

FUNCTION Region_Of (B : Boundary) RETURN Region;
  BEGIN
    RETURN B.Base_Region;
  END Region_Of;
```

```
FUNCTION Next_Pat (B : Boundary;
                   A : Adjacency;
                   D : Direction;
                   C_On, C_previous : Coordinates)
     RETURN Coordinates is
    I, J, K, N : Index;
    Found : Boolean;
  BEGIN
    IF (B.Boundary_Members(C_On.Row, C_On.Column) = Off)
       OR (B.Boundary_members
              (C_Previous.Row, C_Previous.Column) = Off)
       OR NOT (Adjacent (C_On, C_Previous, A))
      THEN RAISE Incompatible_Coordinates;
      ENDIF;
    K := C_On.Row - C_Previous.Row;
    N := C_On.Column - C.Previous.Column;
    Found := False;
    WHILE NOT Found LOOP
      CASE K is
        WHEN -1 => IF D = Clockwise
                     THEN N := N + 1;
                     ELSE N := N - 1; --Counterclockwise
                     ENDIF;
        WHEN  0 => IF D = Clockwise
                     THEK K := K + N;
                     ELSE K := K - N; --Counterclockwise
                     ENDIF;
        WHEN +1 => IF D = Clockwise
                     THEN N := N - 1;
                     ELSE N := N + 1; --Counterclockwise
                     ENDIF;
        WHEN OTHERS => RAISE Incompatible_Coordinates;
        END CASE;
      IF Abs (N) = 2
        THEN N := N/2;  K := 0;
        ENDIF;
      I := C_On.Row + N;  J := C_Column + K;
-- Have the next 8-neighbor to try
      IF (I IN RANGE R.Row_First..R>Row_Last)
         AND (J IN RANGE R.Column_First..R.Column_Last)
         AND THEN (B.Boundary_Members (I, J) /= Off)
         AND THEN (Adjacent ( (I, J), C_On, A))
        THEN Found := True;
      ELSEIF (I = C_On.Row - C_Previous.Row)
             AND (J = C_On.Column - C_Previous.Column)
          THEN EXIT LOOP;
          ENDIF;
        END LOOP;
       IF Found
        THEN RETURN (I, J);
        ELSE RETURN C_On;
        ENDIF;
       END Next_Pat;
```

BIOGRAPHICAL SKETCH

Robert C. Hood was born in Midwest City, Oklahoma, on November 14, 1952. He lived his childhood years in Cocoa, Florida, near Cape Canaveral during the time when man first began to explore the space beyond Earth's atmosphere. He is a summa cum laude graduate of Cocoa High School (1970) and a distinguished graduate of the United States Air Force Academy (1974) where he majored in physics and math. He is an experimental test pilot for the U. S. Air Force and a distinguished graduate of their Test Pilot School (1981). He has over 2500 flight hours in over 30 different aircraft and has worked on several aircraft development programs. In 1983 he received a Master of Science degree from the University of Florida Computer and Information Sciences Department, College of Engineering.

Bob is currently testing the operational capabilty of new aircraft at the Air Force Operational Test and Evaluation Center in Albequerque, New Mexico.